MASTER IN COMPUTER SCIENCE

Implementation of a pure Mercury debugger

Annet, Olivier

Award date:
2008

Awarding institution:
University of Namur

Institut d'informatique
Facultés Universitaires Notre-Dame de la Paix
21, rue Grandgagnage
B-5000 Namur
Belgium

Implementation of a pure Mercury debugger

Olivier Annet

Under the supervison of W. Vanhoof and P. Ross

Thesis submitted for the degree of
Master of Computer Science in the University of Namur

⟅ June, 2008 ⟆

Abstract

Mercury is a relatively new purely declarative logic programming language designed to provide the support that groups of application programmers need when building large programs. Most common errors of classic programming languages are detected by the Mercury compiler thanks to its strong type, mode and determinism system. However, the compiler can not catch, for example, logical errors. In this way, debuggers help programmers to identify bugs that would take hours to find them manually.

This thesis presents the implementation of a pure Mercury debugger. Mercury has different back-ends, which means that the Mercury compiler compiles the Mercury code into another language (C, Java, Erlang, etc.). Contrary to the previous Mercury debuggers, this new one operates at another level allowing this debugger to work independently from the back-end.

Résumé

Mercury est un langage de programmation logique purement déclaratif relativement nouveau et conçu pour fournir le support nécessaire aux groupes de programmeurs d'applications lorsqu'ils conçoivent de larges programmes. La plupart des erreurs communes des langages de programmation classique sont ici détectées par le compilateur grâce à son système exigent de typage, de mode et de déterminisme. Cependant, le compilateur ne peut pas trouver, par exemple, les erreurs de logique. Ainsi, les debuggers sont parfois utiles pour aider les programmeurs à rapidement identifier les bugs de programmation qui prendraient des heures s'il fallait les trouver manuellement.

Ce mémoire présente l'implémentation d'un debugger purement en Mercury. Mercury possde différent back-ends, que le compilateur Mercury utilise pour compiler le code Mercury dans un autre langage (C, Java, Erlang, etc.). Contrairement aux debuggers Mercury précédent, ce nouveau debugger est utilis à un autre niveau lui permettant de travailler indépendamment du back-end utilisé.

Acknowledgements

It is a pleasure to thank the many people who made this thesis possible.

It is difficult to overstate my gratitude to Peter Ross, my supervisor in Australia who helped me to discover and appreciate the Aussies hospitality, custom, food and places. I thank him, Ian MacLarty and especially Peter Wang, for their tremendous help during my work at MissionCritial Pty Ltd. Throughout my internship period, they have provided encouragements, sound advices, good teaching, good company, and lots of good ideas. I would have been lost without them.

This work would not have been possible without the support and encouragement of Zoltan Somogyi and the Mercury Team at the University of Melbourne for all their remarks. Discovering and working on Mercury was a real pleasure.

I wish to thank Wim Vanhoof for all his patience, help for corrections and who gave me great advices. Thank to Didier and Christine Annet for their precious help for the thesis finalization.

Lastly, and most importantly, I wish to thank my parents, Anne-Marie and Pierre Annet. They supported me and make this journey in Australia possible. To them I dedicate this thesis.

Olivier Annet

Namur, May 2008

To the memory of my brother.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Almost all software contains bugs. These are defects in a program which cause it to behave in a non intended manner. There are various types of bugs, some are benign (such as a word badly written) or they can be catastrophic and cost millions of dollars. There are numerous examples and for a concrete one, read Ariane 5 disaster in [1].

Since many years, a lot of programming techniques have helped developers to reduce bugs in software systems. Code reviews and testing certainly helps to reduce the number of bugs in a system as do applying design patterns such as control abstraction and data abstraction. It is common to apply formal verification to mathematically demonstrate the correctness of algorithms underlying a system with respect to a certain formal specification.

However, even with a valid mathematical demonstration, it is often considered as impossible to write completely bug-free software which has a real complexity. So bugs are categorized by severity, and low-severity non-critical bugs are generally tolerated, as they do not impact the proper operation of the system, for the majority of users.

Debugging is the methodical process which detects and reduces the number of bugs, or defects, in a computer program (or a piece of electronic hardware) thus making it behaves as expected.

Usually, many programmers consider that debugging is a frustrating and unproductive activity [2]. Debugging is a tedious task and requires considerable manpower. Indeed, the most difficult part of debugging is to locate the erroneous part in the source code once a symptom is observed (a symptom is an incorrect behaviour of the program). Once a mistake is found, correcting it is usually easy. For small programs, bugs are revealed simply by reading through the code with a critical eye. Another method might be the usage of extra statements that are logging the state of variables at different key points in the program, also called "printf" debugging. It can be very time consuming as the program must be recompiled and re-executed each time that the programmer wishes to see the state of the program. The extra statements make the program less readable, and also increase the risk

that the programmer forgets to remove all logging statements before releasing the software. For large software, developed by teams of programmers, this approach becomes impractical.

Tools exist to help programmers to locate bugs without them having to modify the source code. These tools are known as debuggers. However, even with these tools, locating bugs is sometimes an art; when various subsystems are tightly coupled, it is not uncommon to find a bug in a section of the program that cause failures in a different section, thus making it especially difficult to track.

Typically, the first step in pinpointing a bug is to find a way to reproduce it easily. Once reproduced, the programmer can use a debugger to monitor the execution of the program, see what is going on in the faulty region, and find the point at which the program went astray. Small errors are corrected by a simple modification of the line of code. But, a logical error is an error of thinking or planning from the programmer. Such mistakes require sometimes a whole section of the program to be rewritten. It is not always easy to reproduce bugs. Some bugs are triggered by inputs to the program which may be difficult or impossible for the programmer to reproduce (like a network state for example).

Tracing debuggers, such as the Mercury debugger 'mdb', allow the programmer to view the state of the program at any point during its execution and quickly jump to a point of interest by skipping previous parts which are not of interest. But debugging can become extremely difficult because the user has to direct the search manually (by a sequence of forward and browsing commands). To do this, the user must have an idea of where the bug might be. These kinds of debuggers show what the program is doing, and not where the bug is.

Another class of debuggers are declarative debuggers. They automate the scientific approach. The scientific approach to find a bug is as follows. The debugger formulates a hypothesis, tests it, and then formulates a new hypothesis based on the validity of the previous hypothesis. Each answer reduces the set of possible cause of bug symptoms. Eventually, if there is only one explanation left, it must be the cause of the bug symptom.

To test the validity of a hypothesis, the debugger asks to an oracle, typically the programmer. If the oracle asserts that the computation is correct, then the bug must be in a subcomputation outside the one the hypothesis was about. If the oracle asserts that the subcomputation is incorrect, then the bug must be in the subcomputation the hypothesis was about or one of its descendant subcomputations, so the next hypothesis is about one of these. This ability to isolate bugs is what distinguishes declarative debuggers from other type of debugging tools.

Since this scientific approach is guaranteed to find the bug if the hypothesis test is reliable, declarative debuggers have the potential to make the bug location task more predictable and therefore cheaper in terms of time and money.

The Mercury language, created by the University of Melbourne, is a purely declarative logic and functional programming language intended to support the creation of large, reliable programs. The Mercury compiler is under continual development and contains more than a hundred thousand lines of code is a perfect test case.

The compiler heavily transforms the Mercury programs before any code generation. Procedures are put into superhomogeneous form (Read Chapter 3: Mercury), higher order predicates are transformed into first order predicates; under the commutative semantics, conjunctions and disjunctions can be evaluated in any order. Hence, it is sometimes difficult to relate Mercury source code with its corresponding trace, even if the program is compiled using the strict sequential semantics with all the optimizations switched off. In order to relate the trace with the initial program, it might sometimes be helpful to have a look at the HLDS code [3]. The HLDS for High-Level Data Structure are the compiled program in different optimization states.

Mercury uses different backends. A backend is the targeted language in which that the Mercury code will be transformed into. It could be low or high-level C code, Java, .NET or Erlang.

Currently Mercury only has a debugger which works for one backend, the low-level C. Relating this code with the Mercury code can be difficult at times, so disposing of a pure Mercury debugger using another backend (here the high-level C code or the Erlang), could be helpful. That is what this research aims to do. The debugger has been implemented via the source-to-source transformation described in the paper [4]. In the following, the new debugger will wear the name of source-to-source debugger or 'ssdb'.

The remainder of this thesis has been divided up as follows.

In chapter 2 we introduce a description of MISSION CRITICAL Europe SA and its branch MISSION CRITICAL Australia Pty Ltd, the company of my internship. These companies use logic programming and ontology for real world applications.

In chapter 3 we give the background which will be needed for the rest of the thesis. We introduce the Mercury programming language followed by our first execution of a basic program with the compiler.

In chapter 4 we describe the previous work on debuggers. We observe what exactly a debugger is, and we detail the current procedural Mercury debugger. Then we proceed with the related declarative Mercury debugger with its bugs search algorithms.

In chapter 5 we present the current work and state of the source-to-source debugger. We introduce its concepts and its theoretical basis structure based on the paper [4]. Then

the point on the practical basis structure proposes some modifications to the debugger due to conflicts between theory (in paper [4]) and the conception of the Mercury execution system. And we finally summarise the different commands of the debugger. To conclude, before the conclusion, we shall see some important directions for further work on the source-to-source debugger. All these improvements are, most of the time, already present in other debuggers and have to be implemented in addition to current features to put the source-to-source debugger to the same level as its alter ego.

Chapter 2

Mission Critical

In a world where computers are becoming ubiquitous, the development of reliable, cost effective and maintainable computer software is a real challenge. The problem today that computer scientists must solve is to find a new method to build reliable software, especially considering the rapid changes of user requirements for their software [5, 6]. Mission Critical Australia Ltd Pty (MC) is a company devoted to providing customers with a solution to this challenge. Headquartered in Brussels, Belgium, MC also has a subsidiary in Melbourne, Australia and a branch is about to be created in Vancouver, Canada. The objectif of MC is to be able to work 24h/24. It employs 15 qualified computer scientists spread over the two current sites.

MC has pioneered an approach which allows one to build reliable software which is flexible as requirements change. The key to this approach is to describe the problem to be solved in a formal way by the people who understand the problem and then to consume that description directly in the software that is written.

Ontology is used to formally describe the problem domain. Ontology allows someone to describe concepts, the relationship between concepts, instances of concepts and the relationship between instances. For example, Person and Uncle are concepts. The relationship between Persons and Uncles is that an Uncle is a Person who has a sibling who has children. Similarly we can define the instances of Person, Alice and Bill. Alice and Bill are related to each other by the fact that they are siblings. The W3C have standardized an ontology language, OWL, that MC uses to describe problem domains [7].

This approach has enabled MC to successfully develop highly complex operational applications within a defined budget and timeframe. This approach is known as ODASE, Ontology Driven Architecture for Software Engineering.

ODASE is based on the following principles:

- It is the **Business** who possesses the necessary experts which have knowledge and will help engineers to build Ontology of the business problem. This Ontology is a model defining unambiguously the business concepts, relationships and business rules using standards defined by the W3C.
- **IT** experts use ODASE robots and generate, from this Ontology, the corresponding program source code, which is then compiled into a high performing and scalable application.

With ODASE, the Business-IT gap is bridged by describing the business knowledge in a formal language understandable by Business and IT experts, and consumable by computer programs. When adaptive maintenance is required, the ontology is changed and new code is generated [8].

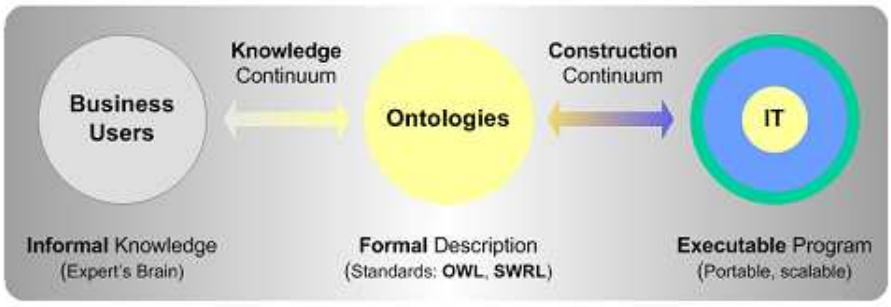

Figure 2.1: Bridge between the Business and the IT made by ontologies

This tight integration of business model and code delivers low cost, high flexibility, swift time-to-market and great usability for business-critical Internet software and allows Mission Critical IT to offer an explicit warranty on the software, a unique proposition [8].

The robots that MC has developed generates a type and mode safe interface to the OWL Ontology in the programming language, Mercury. Mercury has a formal semantics which can be used to map to the OWL semantics allowing one to consume the OWL Ontology directly in the executable code. Thus having a logical continuum all the way from the user's definition of the problem domain to the executable code.

This is how MC can achieve 'zero defects' software.

Chapter 3

Mercury

Logic programming languages are theoretically superior to imperative programming languages such as Pascal, C, C++, and Ada because they operate on a higher level. In this way, for example, it is impossible to use a variable that has not been initialized or get the so frustrating "pointer segmentation fault" error such as C.

Mercury combines the clarity and expressiveness of declarative programming with advanced static analysis and error detection features. Declarative means that we write what we want as result rather than how to get this result. Its highly optimised execution algorithm delivers efficiency similar to conventional programming systems. Mercury addresses the problems of large-scale program development, allowing modularity, separate compilation, and numerous optimisation/time trades-offs.

Mercury is syntactically similar to Prolog with some extra declarations. However, Mercury is semantically very different from Prolog. Mercury is a pure logic programming language with a well-defined declarative semantics. It provides declarative replacements for Prolog 's non-logical features and it does not retain any non-logical features; in Mercury even I/O is declarative [9].

Moreover, the Mercury compiler catches most of the common errors thanks to its strong type, mode and determinism system. Many bugs that would be classified logic errors in imperative or in Prolog turn out in Mercury to be mode errors or determinism errors that can be detected by the compiler, leaving only a small minority of real logic errors to chase down using manual debugging methods. This system also allows more profound optimizations and it results in a significant improvement in speed in comparison with Prolog or other similar languages.

As stated in [10]: *"Mercury is designed to appeal to at least two groups of programmers. One group is those with backgrounds in imperative languages such as C who are looking for a higher level and more expressive language. The other group is those with backgrounds in logic programming languages such as Prolog who are looking for a genuinely declarative language that supports the creation of efficient and reliable software solutions to large and complex problems."*.

The Mercury features are examined and resumed in the following pages. These features are the syntax, the type, the mode and the determinism of Mercury. The final section of this chapter is devoted to the Mercury compiler.

3.1 Mercury syntax and execution overview

A logic program is a representation of a domain for which we have to solve a problem. Domain knowledge is represented by a set of predicates and functions. The basic building blocks are **atoms** and **terms**.

Terms are defined inductively in the following way [11]:

- a variable is a **term**
- a constant is a **term**
- if f/n is a functor of arity n
  $t_1, ..., t_n$ are terms
  then $f(t_1, ..., t_n)$ is a term

Atoms (or *well-formed formula*) are defined in the following way [11]:

- if p/n is a predicate of arity n
  if $t_1, ..., t_n$ are terms
  then $p(t_1, ..., t_n)$ is an atom

Predicates and functions are represented either by facts and rules. A fact is an assumption or a basic relation on the domain. A rule determines a relation between different relations of the domain. In other words, a fact is a particular case of a rule. Terms are object used by these facts and rules. [12, 13, 14]

- A **function fact** is an item of the form '*Head = Result*'.
- A **predicate fact** is an item of the form '*Head*'.
- A **function rule** is an item of the form '*Head = Result :- Body*'.
- A **predicate rule** is an item of the form '*Head :- Body*'.

When '*Head*' denotes an atom and '*Body*' a goal.

In the following pages, predicates (or functions) will be represented by the notation '*name/arity*', such as '*main/2*'; it means that the predicate '*main*' has two arguments.

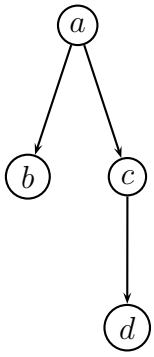

Figure 3.1: Representation of arcs and nodes problem

Example 3.1.1 *The graph of Figure 3.1 can be represented by the following facts defining a predicate arc/2.*

```
arc(a,b).                    (I)
arc(a,c).                    (II)
arc(c,d).                    (III)
```

The predicate *path/3*, where *path(X,Y,L)* denotes that there is a path L from X to Y, can be defined as follows:

Example 3.1.2 *Rules to find a path in the Figure 3.1.*

```
path(X,Y,[arc(X,Y)]) :- arc(X,Y).            (1)
path(X,Y,[arc(X,Z)|L]) :- arc(X,Z), path(Z,Y,L).    (2)
```

In Mercury, the body of **rules** are represented by *goals*. The most common types of goals are conjunction, disjunction, if-then-else, unification and procedure call. In the above example, the body of the second rule,

```
arc(X,Z), path(Z,Y,L)
```

is a conjunction with two conjuncts. Intuitively, the rule states that, for any variables X and Y,

```
path(X,Y,[arc(X,Z)|L])
```

will be true if both conjuncts in the body are true. Refer to [12, 15] for a full description of these concepts.

The user interacts with the domain knowledge expressed by a logic program by means of a query. A query is an existential question that we would like to be answered by the logic programming system. For example, If we want to know if there is a *path* between the node 'a' and 'd', we can pose the query:

```
?- path(a,d,L).
```

The execution of a logic program is like a theorem proof. The execution algorithm uses facts and rules to resolve the query. This can be formalized by the construction of an SLD-tree which represents all solutions that can be built upon logic axioms. A node in a SLD-tree represents the state of a derivation. A node can have one or several child nodes. Each child represents the unification of the parent node with the head of a clause. Unification is a binding of the variables within two terms or atoms. Unification is possible if a substitution makes the terms identical. A substitution θ is called a *unifier* for the set of simple expressions S if Sθ is a singleton. A unifier θ for S is called a *most general unifier* (*mgu*) for S if, for each unifier σ of S, there exists a substitution γ such that $\sigma = \theta\sigma$ [11]. A path in the SLD-tree is a derivation. Leafs in the tree are either *fail* or *true*. A *fail* denotes that the parent does not unify with any clause. A *true* denotes an empty query that is there are no more atoms to resolve. A derivation ending in *true* represents a refutation and hence a success. Each refutation has an associated computed answer, which is the comparison of the mgu's computed along the branch. The Figure 3.2 represents an SLD-tree for the query *path(a,d,L)* and the program in Example 3.1.2.

Most languages use a depth-first search and backtrack mechanism with a leftmost atom selection to construct the SLD-tree and hence to solve the query. In other words, we continue on the same branch until completion (succeeds or fails) before trying new ones. When a path fails, the algorithm backtracks to the last decision point which is a decision point where we could have chosen a different clause for unification, and then try an unexplored alternative. If a node represents a conjunction, then the leftmost atom is chosen. In the example, the atom selected is underlined. However, the atom selection is up to the language and so is the selection of the clause which we unify the atom to.

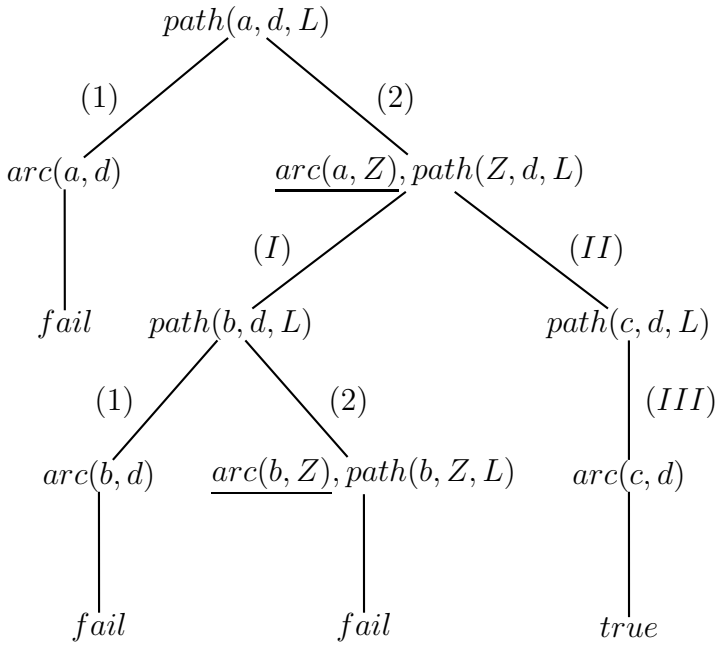

Figure 3.2: SLD tree from Example 3.1.2

The resolution of the problem in Figure 3.2 goes as follows: Let us consider the definition of *path/3* given before to resolve the query *path(a,d,L)*. The root of the tree has two child nodes. The first one result from the unification of the root atom with the head of the rule (1) *path(X,Y,[arc(X,Y)]) :- arc(X,Y)*. Terms 'a' and 'd' unifie respectively with 'X' and 'Y' and we get *arc(a,d)*, that is a new query constituted of the body of the clause on which the previously computed mgu is applied . The new query does not match with any head declared, so it fails. We backtrack to *path(a,d,L)* and use the second rule, number (2) *path(X,Y,[arc(X,Z)|L]) :- arc(X,Z), path(Z,Y,L)*, which creates a new branch. We decide to consider the leftmost atom for unification, this because we use a leftmost atom selection. Then, by the first rule, there is a matching with two facts. *arc(a,Z)* can be unified with the fact (I) *arc(a,b)* or (II) *arc(a,c)*. If we consider the unification with (I), only *path(b,d,L)* becomes relevant for the success of the clause. We can see that this atom does not match with any (head of) clause. If the clause is unified with rule (2), no fact will match with *arc(b,Z)* and thus fails. In this case, we backtrack to the clause *arc(a,Z), path(Z,d,L)* and we unify the leftmost atom with the rule (1) and the fact (II), so we bind 'Z' with 'c'. In this case, only *path(c,d)* becomes relevant and it succeeds by matching with the rule (1) and the fact (III) *arc(c,d)*. The fact that we have constructed a derivation resulting in success resolves our query. There is a path from node 'a' to the node 'd ' by 'c'.

```
L = [arc(a,c), arc(c,d)].
```

The following is the Example 3.1.2 translated in Mercury

Example 3.1.3 *arc and path predicates written in Mercury*

```
1  :- module find_path.
2  :- interface
3  :- import_module io.
4
5  :- pred main(io::di, uo::di) is det.
6
7  :- implementation.
8  :- import_module string.
9
10 main(!IO) :-
11       path(a,d,L),
12       io.write(L, !IO),
13       io.nl(!IO).
14
15
16 :- pred arc(string, string).
17 :- mode arc(in, in) is det.
18
19 arc(a,b).
20 arc(a,c).
21 arc(c,d).
22
23 :- pred path(string, string, list(T)).
24 :- mode path(in, in, out) is semidet.
25 :- mode path(out, out, in) is det.
26
27 path(X, Y, [arc(X,Y)]) :- arc(X, Y).
28 path(X, Y, [arc(X,Z)|L]) :- arc(X, Z), path(Z,Y,L).
```

Let's analyse each part of this program [3].

The first lines of a Mercury program are often like this:

```
:- module find_path.
:- interface
:- import_module io.

:- pred main(io::di, uo::di) is det.

:- implementation.
:- import_module string.
```

It declares the content of the module '*find_path*' which is saved in the '*find_path.m*' file. The *interface* code lines describe all visible construction (predicates, functions, etc.) to other modules that might import '*find_path*'. Later, we have the *implementation* section, which is hidden to other modules. Then we have:

```
:- pred main(io::di, uo::di) is det.
```

This line declares the signature of the '*main*' predicate and illustrates three strong points of Mercury, namely the use of type, mode and determinism information.

Both parameters are of the type '*io*', which are used to read and write data on standard streams (read from a file, write data to the screen, etc.). The type concept is explained later in section 3.2. The first argument has the mode '*di*' for *destructive input*, and the second has the mode '*uo*' for *unique output*. Both of these modes and others are explained later in details in section 3.3. Finally, we got the '*det*' word for the determinism. It means that the predicate is deterministic and will have *exactly one solution*. All determinisms are described in section 3.4.

If we look to the body of the predicate *main/2* , we have:

```
main(!IO) :-
    path(a,d,L),
    io.write(L, !IO),
    io.nl(!IO).
```

It calls the *path/3* predicate and writes the solution on the standard output stream with the *io.write/3* predicate from the '*io*' module:

```
:- pred io.write(T::in, io::di, io::uo) is det.
```

The *path/3* predicate has the following signature and code:

```
:- pred path(string, string, list(T)).
:- mode path(in, in, out) is semidet.
:- mode path(out, out, in) is semidet.

path(X, Y, [arc(X,Y)]) :- arc(X, Y).
path(X, Y, [arc(X,Z)|L]) :- arc(X, Z), path(Z,Y,L).
```

The two different modes of the predicate *path/3* represent two different usages of the predicate. More details will be given about type, mode and determinism in the following sections.

3.2 Types

As most other languages, Mercury has some basic built-in types: int, float, string and char. It is also possible to construct new types including arrays, sets and lists by means of discriminated union and equivalence types [16]. Finally, Mercury has two special types respectively *univ* to represent any other type or *io.state* to perform I/O [12]. We are going to see them in this order.

The discriminated union type. It is a description of a possibly infinite set of values by a finite number of constructors. Each constructor can have its own specific arguments.

Any discriminated union looks like this:

Example 3.2.1 *Some discriminated type example*

```
:- type event_type
    --->    call
    ;       exit
    ;       fail
    ;       redo.

:- type list(int)
    --->    []                  % The empty list
    ;       [int | list(int)].
```

In the first case, an *event_type* can take one of the four ports (*call, exit, fail or redo*). In the second case, the type *list(int)* can be either an empty list or the possible infinite list with one integer followed by a *list(int)*.

Example 3.2.2 *Instantiation examples of type list(int)*

```
[]
[1,2,3]
[1,1,2,3,5]
```

Both constructors of the *list(int)* might not be confused with each other.

The type *list(int)* is a list of integers. Its definition may be generalized by the following construction:

```
:- type list(T)
    --->    []                  % The empty list
    ;       [T | list(T)].
```

In this way, the type *list(T)* might take any argument type.

Each field in the constructor can be named and individual fields can be accessed for within a program. Furthermore, constructors may be overloaded. In other word, they can have multiple occurrences having the same name and arity, as long as they all have different argument types.

The equivalent type. Equivalence identifies one type name with another. For example:

Example 3.2.3 *An equivalent type*

```
:- type position == int.
:- type list_position = list(position).
```

In this way, *list_position* have to be used as a list of integers and have the same features.

The "universal" type: univ. The type univ is used when one needs to transform different types in a homogeneous structure. It is possible to perform this by using the predicates *type_to_univ*/2 and *univ_to_type*/2, which convert any type to the universal type and back again.

The "state of the world" type : io.state. When a predicate uses an I/O, the old state of the world is passed and produces a new state of the world. Like names in mathematics, once bound, a logical variable cannot change that binding afterwards. Here is an example with an io.state variable and the predicate *write/3*:

```
io.write(L, !.IO_In, !:IO_Out)
```

Both value '*!.X*' and '*!:X*' represent respectively the current and the next value in a sequence labelled X. Note that both of these values may be represented by a syntactic sugar of the form '*!X*'. That is, '*p(..., !X, ...)*' is parsed as '*p(..., !.X, !:X, ...)*' [12].

This is how Mercury gives a declarative semantics to the code that performs I/O.

3.3 Modes

Basically, the mode of a predicate, or function, is a mapping from the initial state of instantiation of the arguments of a predicate, or the arguments and result of a function, to their final state of instantiation [12].

The two most common modes are the standard notion for inputs and outputs. A variable not bound to anything is said to be *free*. Conversely, a variable bound to some piece of data containing any free data is said to be *ground*.

If a variable contains some ground data at the start of the call and, since we can never alter the contents of a bound variable, it must be in the same instantiation state after the call, it is an input variable. Thus, the definition of the in mode is:

```
:- mode in :: (ground -> ground).
```

An output variable, however, is one that is expected to be free at the start of the call and which will contain some ground data at the end of the call. The mode definition of *out* is therefore:

```
:- mode out :: (free -> ground).
```

The mode can be supplied in the pred declaration or as a separate declaration. So, the *path/3* predicate view before can be declared in the following way:

Example 3.3.1 *Merge pred and mode declaration*

```
:- pred path(string::in, string::in, list(T)::out) is semidet.
```

However, it happens often that a predicate has several modes. Each mode represents a different usage and corresponds to a different procedure in the compiled code.

path/3 comes from the Example 3.1.3. It can be specified in Mercury by the following declarations.

Example 3.3.2 *Several modes for a predicate*

```
:- pred path(string, string, list(T)).
:- mode path(in, in, out) is semidet.
:- mode path(out, out, in) is semidet.
```

Each mode declaration corresponds to a different usage of this predicate:

- in the first one, the user provides start and end point and wishes compute a path between them.
- in the second one, the user provides a path and wishes to verify that the path exists and to compute begin and end point.

Finally, there are some other modes available, called *unique modes*. In Mercury, the concept of *uniqueness* which say that an object is unique if there is only one reference to it at any point in the computation. There are three standard modes for manipulating unique values:

```
:- mode di == unique >> dead.    % destructive input
:- mode uo == free >> unique.    % unique output
:- mode ui == unique >> unique.  % unique input
```

Unique means that the value has only one reference. Whereas the *dead* means that there is no more reference to this value. Mode '*uo*' is used to create a unique value. Mode '*ui*' is used to inspect a unique value without losing its uniqueness. Mode 'di' is used to desallocate or to reuse the memory occupied by a value that will not be used. The two most used are '*di*' and '*uo*' mode by using IO. For example, the previous *write/3* predicate has the following signature:

```
:- pred io.write(T::in, io::di, io::uo) is det.
```

3.4 Determinism

Determinism declarations are attached to mode declarations. The programmer and the compiler analyse the bodies of procedures to categorise these modes according to the maximum number of solutions it can produce (zero, one, or more than one) and whether or not it can fail before producing its first solution [17]. The determinism of goals is inferred from the determinism of their component parts. Thus the inferred determinism of a procedure is just the inferred determinism of the procedure body. In a short form [12]:

	Maximum number of solutions		
Can fail?	0	1	>1
no	erroneous	det cc_multi	multi
yes	failure	semidet cc_nondet	nondet

Table 3.1: Mercury's determinism

The annexe 2 presents five types of determinism with an example.

The four determinisms det, semidet, multi and nondet suffice for the vast majority of procedures but there are a very small number of procedures which require another determinism because they never produce a solution. The determinism failure is used for procedures that always fail. Computations that always fail may not export any binding, and can be replaced by the goal fail. The determinism erroneous is used for procedures that operationally correspond to aborting execution; from program's point of view they neither succeed nor fail. (The logical semantics for erroneous procedure is that they loop, e.g., Evaluated to undefined.) [17].

As seen above in the determinism annotations described earlier, there are '*committed choice*' versions of multi and nondet, called *cc_multi* and *cc_nondet*. These is typically used instead of multi or nondet if all calls to that mode of the predicate (or function) occur in a context in which one, and *only one solution* is needed, whatever this solution is. We call it a single-solution context. There are several efficiency reasons to use committed choice determinism annotations. See [12] for more details.

3.5 Higher order terms

Mercury supports higher-order predicates; which is a predicate passed as an argument to another predicate. The higher order term appears as a variable in the procedure and its utilisation is similar to that of a regular predicate. It can be called by placing arguments in parenthesis after the higher order variable.

For example, a '*map*' predicate, which applies a higher order term to the elements of a list to produce a new list, could be defined as follows:

Example 3.5.1 *map/3 predicate*

```
map(_, [], []).
map(P, [H0 | T0], [H | T]) :-
    P(H0, H),
    map(P, T0, T).
```

Actually, one can pass only a single mode of a predicate as a higher-order argument. To pass a whole predicate in argument, one has to pass multiple higher-order procedure terms.

3.6 Compiler

The Mercury compiler is called 'mmc' for Melbourne Mercury Compiler. It uses different back-ends. Each of them generates a specified code: high- and low-level C, Java, .NET, Assembler or Mercury byte-code. The most commonly used is the low-level C back-end. The low-level C is fast and portable under almost all software and hardware platforms. See annexe 1 for details on 'How to use the Mercury compiler'.

Mercury uses *eager* evaluation, also known as *strict* evaluation, meaning that an expression is evaluated as soon as it gets bound to a variable. It is in contrast with *lazy* evaluation, or *non strict* evaluation, meaning that a computation is delayed until the result of the computation is known to be needed [18, 19].

Besides, Mercury programs are heavily transformed before any code is generated. The program goes through a set of passes during compilation. Once all of them are executed, the program is compiled. Mercury's type, mode, and determinism verification systems are three of these passes and ensure that many of the most trivial programming errors are caught at compile-time rather than at run-time. Procedures are put into superhomogeneous form, high order predicates are transformed into first order predicates. All of these passes speed up, on one hand, the development of applications by catching most of errors before the execution; and on the other hand, the execution of these applications thanks to the optimizations from the compiler. In fact, as most of information is given by the user, the inference engine does not have to compute it itself.

The program is compiled using other high-level optimizations, including automatic inlining, common subexpression elimination, predicate specialization to eliminate unused arguments, specialization of higher-order predicates when their higher-order arguments are known, and the reuse of the storage of terms that occur more than once in a predicate. Finally, the compiler also implements a whole host of low-level optimizations. These include constructing ground terms at compile time, stack slot allocation using graph colouring, elimination of dead labels and code, and many others [3]. Read [20] for a detailed description of optimizations operated by the Mercury compiler.

Chapter 4

Previous works on debuggers

In this part of my thesis, I am considering that it is important to mention, describe and resume some of the works done before me in the same domain. Obviously, substantial parts of this section are based on [21, 22, 2, 23, 24, 25].

Debugging is, in general, a tiring task due, mostly, to the size of the program or the limitations of the debuggers. The difficulty of software debugging varies greatly with the programming language used and the available tools, such as debuggers.

In lower-level languages such as C or assembly, it is often difficult to detect bugs that may cause silent problems, such as memory corruption, and pinpoint the initial problem might take hours. In those cases, memory debugger tools may be needed. Debuggers such as gdb, lcc and cdb are quite old in the domain for debugging C code, see the documentation on how to debug a C/C++ program in [26, 27, 28, 29].

Generally, the debugging of high-level programming languages, such as Java, is easier because they have features such as exception handling that make real sources of devious behaviour easier to spot. Java debuggers usually implement thread debugging, which is not yet available on Mercury debuggers. Do not hesitate to see [30, 31] for full details on how-to-use a java debugger.

In specific situations, *static code analysis tools* can be very useful. These tools identify complex programming bugs that can result in system crashes, memory corruption, and other serious problems within the source code. Some issues detected by these tools would take weeks to identify with a traditional compiler or interpreter, since these tools generally perform a more thorough and complex semantic analysis than a classic compiler. Both commercial and free tools exist in various languages. These tools can be extremely useful when checking very large source trees, where it is impossible to perform code walkthroughs. However, these tools have always a various rate of false positives bugs detected. *Splint* (for C) or *Bandera* (for Java) are examples of free software. Commercial tools like *CAST* can analyse more than twenty-five languages [21].

Similar tools exist for debugging electronic hardware (e.g., computer hardware) as well as low-level software (e.g., BIOSes, device drivers) and firmware. Instruments such as oscilloscopes, logic analyzers or incircuit emulators (ICEs) are often used, alone or in combination.

Basic debuggers exist also for logic programming languages. Opium (for Prolog [32]) and Opium-M (for Mercury [33]) or Morphine (for Mercury [34]) are some well-known examples. But all of these examples are specialised in a specific domain, they are specific for a back-end and interact at a very low-level with the runtime system. This is why MC asked me to realize another type of debugger. Firstly, this chapter explains in details what a debugger is. Then, secondly, it presents two different types of debuggers for Mercury. And finally, the next chapter introduces the new debugger that I did during my internship.

4.1 What is a debugger?

Debuggers enable the programmer to monitor the execution of a program, stop it, re-start it, set breakpoints, change values in memory and even, in some cases, go back in time (with some limits) avoiding the need to restart debugging from the beginning due to an erroneous input during debugging. Basically, a debugger is an application able to stop the execution of a program at some points, and let the programmer examine the current state of the program: stacks, registers, variables, arguments, and so on. This can give a way for the programmer to hunt bugs in the program and to detect exactly where an error could be.

Many programmers find debugging a frustrating and unproductive activity. A typical debugging session goes as follows:

1. They step through the source code upon encountering a procedure call
2. They check the values of a procedure input argument and find them correct.
3. They step over the execution of the procedure in the debugger, regaining control when the procedure returns to its caller.
4. They check the values of a procedure's outputs, and find some of them to be incorrect.

At this point, they know there is an error somewhere in the call tree of the procedure, but they don't yet know precisely where. The natural action would be to re-execute the call and check the values of variables at a selection of program points before the call returns.

Traditional debuggers, such as gdb, can execute the program only in the usual forward direction, and are often unable to help the programmer in an effective way. As it is

impossible to go backwards, there is no way to recover the previous value of an assigned-to-variable, and it is therefore not possible to restore the computation to the state it had before the call. The only choice for the programmer is to restart from scratch, stop its execution at the problematic call, and examine the execution of the call in more detail. This has several problems:

- If the program modifies data it uses as input, then the programmer must restore the program to its initial state before re-execution.
- Reaching the point of the program before the problematic call may take considerable time.
- The programmer may arrive at the call after a long sequence of operations (continue to breakpoint, next etc) and could be unable to repeat it.
- Any deviation from the original input upon re-execution may prevent the debugger from re-establishing the same state of the computation at the time of the call. The only cure is to restart execution one more time.
- If some input comes from sources that are outside the programmer's control, such as network connections, then re-establishing the state of the computation at the time of the call may never be possible.

A mechanism that would allow the debugger to reset the computation to the state it had at the time of the call, effectively allowing the programmer to jump backwards in the program's timeline, would avoid these problems. Problems and solutions for this purpose are described in [35].

4.2 Mercury debuggers

The following section present two different Mercury debuggers. The procedural debugger transforms the code by adding trace events, allowing it to stop the execution at each event. The declarative debugger asks questions to the user to detect which part, in a predicate, is buggy.

4.2.1 Mercury procedural debugger

The Mercury debuggers view the execution of a program as a sequence or *trace* of events; when debugging is enabled, the compiler generates code that gives the runtime system control at each event. The runtime system can then interact with the programmer, allowing him to inspect the state of the computation and to issue commands. Some of these commands tell the debugger to give the control back to the program being debugged, and to interact with the programmer again only at a future event that matches a specified

condition [2].

Events can be classified into two categories, *interface* or *external* events and *internal* events. Interface events describe the interaction between one invocation of a procedure (one mode of a predicate) and its caller, while internal events describe the flow of control inside the call. The four types of interface events supported by the declarative debugger correspond to the four ports in Byrd's box model [25]:

Each call to a procedure is represented by a box. A box represents the invocation of a single procedure. Each box has four ports: they are named the **Call**, **Exit**, **Fail** and **Redo** ports. The labelled arrows indicate the control flow in and out of a box via the ports.

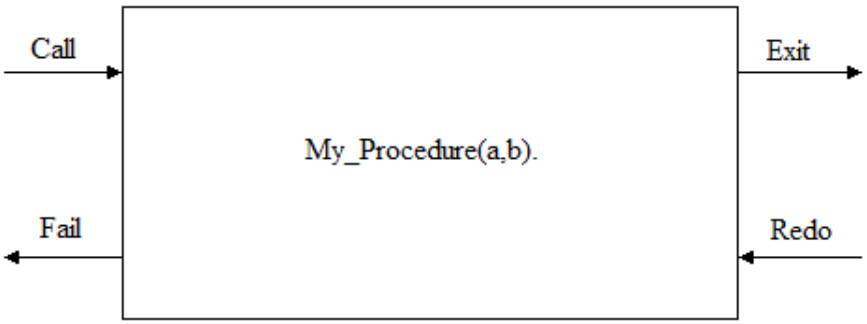

Figure 4.1: The Byrd's box model

If a head of a clause unifies with the goal, then we seek solutions to all the subgoals in the body of the unified clause.

Control then 'flows' into the box associated with the procedure unified through the **Call** port. The **Call** port for an invocation of a procedure represents the first time the solution of the associated goal is sought.

Control reaches the **Exit** port if the procedure succeeds. It only occurs if the initial goal has been unified with the head of one of the procedure's clauses and all of its subgoals have been satisfied. Then control is about to return to its caller.

The **Redo** port can only be reached if the procedure call has been successful and some subsequent goal has failed. Then the system uses *backtracking* to try to find some alterna-

tive way of solving some top-level goal.

Basically, *backtracking* is the way a logical language attempts to find another solution for each procedure that has contributed to the execution up to the point where some procedure fails. This is done back from the failing procedure to the first procedure that can contribute an alternative solution.

During backtracking, control passes through the **Redo** port. The previously unified clause backtracks through the subgoals that were previously unified. We can reach the **Exit** port again if another of its subgoals succeeds, or failing and reach the **Fail** port, then another clause can be unified [21].

In addition of these ports, there is an event used to handle exceptions in Mercury named **excp**.

At each event, the debugger can provide several kinds of information which identify the procedure (the name of the predicate or function, its arity, its mode, its depth, etc.) and the list of its arguments (including their names, types, values, etc).

4.2.2 Mercury declarative debugger

Principes

Logic programming languages have both a declarative semantics and an operational semantics. The declarative semantics views the program as a set of logical statements whose logical consequences give the meaning of a program, while the procedural semantics views the program as a sequence of instructions to be executed by the machine. The declarative semantics is higher level and closer to the specifications and therefore, closer to the way that programmers think during the analysing of the application domain, than the operational semantics. In Mercury, the operational semantics is very closely coupled with the declarative semantics. This makes it much easier to implement a declarative debugger for Mercury, than it would be for a non-strict language, such as Haskell [2].

The fact that Mercury has a clean declarative semantics means that it is amenable to declarative debugging. Declarative debuggers work by asking to an oracle (usually the programmer) questions about the intended meaning of the various parts of the program, and comparing this to its behaviour. By tracking inconsistencies between behaviour and intended meaning, such debuggers can locate a point in the program in which the correct results of a number of subcomputations are combined into an incorrect result. This point is the precise location of a bug in the program source [2]. At this point, we can highlight three parts in the declarative debugger [24]:

- The *backend*, which is responsible for generating the *evaluation dependency tree* or EDT.
- The *analyser*, which searches the EDT for bugs.
- The *oracle*, which gives answer to each query from the analyser to ascertain which nodes are erroneous and which nodes are correct.

It is possible to invoke the declarative debugger from within the procedural debugger in different situations. Depending on the case, the kind of diagnosis that the runtime system asks the algorithm to perform differs [2, 23].

The type of the questions asked by the debugger is roughly:
<node display>
"Is it valid?"

Where <node display> represents some information on the current node in the EDT.

The oracle (the programmer) can answer the questions by either:

yes	the error is not in this node.
no	the error is in this node.
don't know	the oracle does not know if it is *yes* or *no*. The debugger will search for the reply itself by computation or, if it does not find the answer, the debugger will postpone the question.
inadmissible	the answer is neither *yes* nor *no*. The oracle knows that the call should never have happened due, for example, to incorrect inputs.

In any case, any question with an answer different from *don't know* will never be asked again. Other complex commands are possible but they are not described in this thesis, read the declarative debugger part in the *Mercury User's Guide* [3] for details.

In the following points, we will introduce different algorithms used by the declarative debugger to track back a bug in a program.

The declarative debugger general algorithm

The algorithm of the declarative debugger constructs an *evaluation dependency tree* (or EDT). Each node in the EDT corresponds to an **exit**, **fail** or **excp** (for exceptions) event

in the trace. Each of these nodes has an assertion associated: whether the solution represented by an exit event is correct, whether the set of solutions returned before a fail event is complete, or whether the exception thrown at an excp event was expected to be thrown. The declarative debugger searches the EDT for a node in the tree which shows where correct premises are combined together to form an incorrect result; in other words, an incorrect node whose children are all correct represents a bug. Initial methods of declarative debugging constructed a complete EDT and were unusable for real world programming due to their lack of performance. Recent research in logic programming has resulted in different ways to construct the EDT efficiently as explained in [36].

In practice, the EDT is constructed by the Mercury declarative debugger on demand, and subtrees are eliminated with knowledge gained from the oracle. We do not build the entire EDT directly because we need to construct different EDT fragments depending on one hand, the reply from the oracle, and on the other hand, the result of certain constructions during the execution (e.g. negations and if-then-else goals, read [23, 24] for details).

The following schema represents this idea. Each node in the tree corresponds to an exit, fail or excp event. The tree is generated piece by piece and never fully materialized (the debugger constructs only required portions), thanks to information gained from the oracle. By exploring a parent, children nodes are generated below it [22, 2, 23].

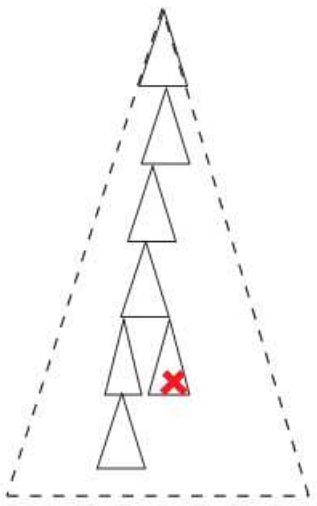

Figure 4.2: SLD tree constructed on demand

The Example 4.2.1 contains the *areas/1* and *area/1* predicates. The *areas/1* takes

a list of shapes in argument and its body uses *area/1* to compute the area of the shape depending of its argument (here, the shape is a box or a circle).

Example 4.2.1 *area and areas predicate example for different search algorithms*

```
area(circle(Radius)) = Radius * pi.        % should be sqr(Radius) * pi
area(box(Width, Height)) = Width * Height.

areas([]) = [].
areas([Shape | Shapes]) = [area(Shape) | areas(Shapes)].
```

The example contains a bug: the computation of the area of a circle is wrong. The objective is to find the bug with the declarative debugger. The questions asked by the declarative debugger depend on the way in which EDT is constructed. We will use the Example 4.2.2 below, which respects the rules from Example 4.2.1, to illustrate a number of different search strategies.

Example 4.2.2 *this line contains a bug.*

```
areas([box(2, 3), box(4, 5), circle(2), box(3, 4)]) = [6, 20, 6.28, 12].
```

Top-down search

Using this mode, the declarative debugger will ask about the children of the last question the user answered '*no*' to. The algorithm asks questions on children in the order they are executed. This makes the search more predictable and easier to understand from the user's point of view as the questions will more or less follow the program execution. The drawback of top-down search is that it may require a lot of questions to be answered before a bug is found, especially with deeply recursive programs. This search mode is used by default when no other mode is specified [22].

The following example represents a session of queries/answers between the declarative debugger and the oracle when debugging the code of Example 4.2.1 with a top-down search.

Example 4.2.3 *top-down algorithm search*

```
areas([box(2, 3), box(4, 5), circle(2), box(3, 4)]) = [6, 20, 6.28, 12].
Valid?
> no

areas(box(4, 5), circle(2), box(3, 4)]) = [20, 6.28, 12].
Valid?
> no
```

```
areas([circle(2), box(3, 4)]) = [6.28, 12].
Valid?
> no

areas([box(3, 4)]) = [12].
Valid?
> yes

area(circle(2)) = 6.28.
Valid?
> no

Found bug:
area(circle(2)) = 6.28
```

Divide and query search

In this search mode, for each question, a node is picked that divides the tree into two roughly equal parts. It result in O(log N) questions on average where N is the number of events between the event where the *declarative debugger* was invoked and the corresponding *call* event. This makes the search possible for long running programs where a top-down search mode would require an unnecessary large number of questions to be answered. However, questions may appear to come from unrelated parts of the program making them harder to answer [22].

Example 4.2.4 *divide and query algorithm search*

```
areas([box(2, 3), box(4, 5), circle(2), box(3, 4)]) = [6, 20, 6.28, 12].
Valid?
> no

areas([circle(2), box(3, 4)]) = [6.28, 12].
Valid?
> no

areas([box(3, 4)]) = [12].
Valid?
> yes

area(circle(2)) = 6.28.
Valid?
> no
```

```
Found bug:
area(circle(2)) = 6.28
```

Subterm dependency tracking

Most of the time, the user wants to see the value of a specific variable. When a call is erroneous, there is often a small part of one of the arguments which is incorrect. If the user indicates that a subterm of an argument is incorrect, the declarative debugger will ask about the call that created the subterm [22].

In the Example 4.2.5, the user is invoking the *browser* which allows him to quickly isolate a term without answering multiple questions from the declarative debugger. From within this browser, the user can access a subterm by specifying what subterms to skip. In the example below, a '2' ignores the term and the '1' isolates the current term to allow the user to examine the details of it.

Example 4.2.5 *subterm dependency tracking algorithm search*

```
areas([box(2, 3), box(4, 5), circle(2), box(3, 4)]) = [6, 20, 6.28, 12].
Valid?
> browse return

browser> cd 2/2/1        % jump to the desired term

browser> print           % print the value of the term
6.28

browser> mark
area(circle(2)) = 6.28   % print the term and its value
Valid?
> no

Found bug:
area(circle(2)) = 6.28
```

The Figure 4.3 shows what exactly does the previous browser command *cd 2/2/1*:

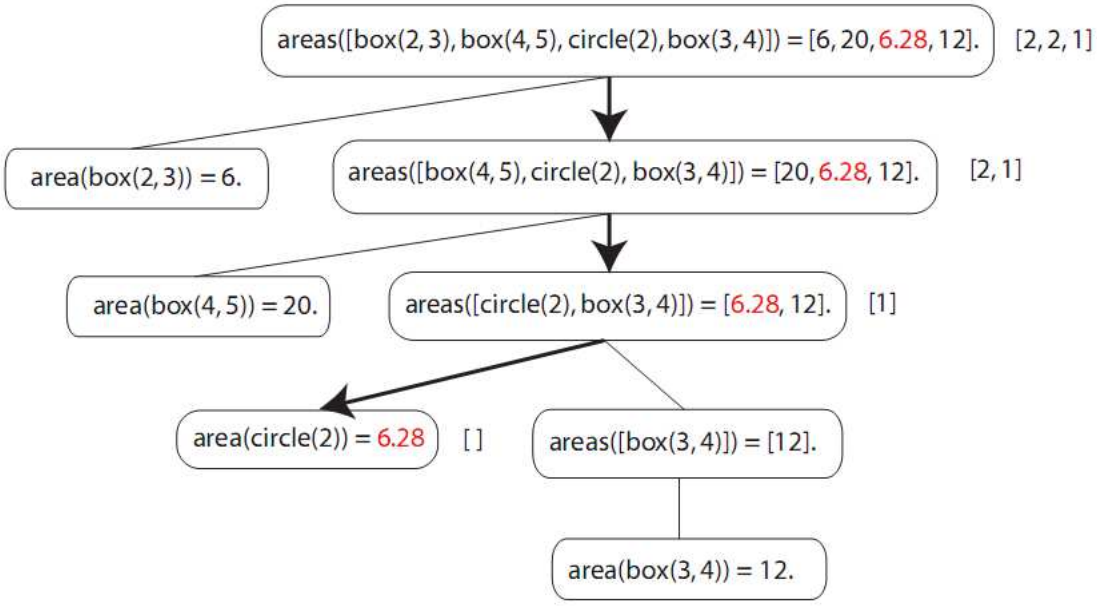

Figure 4.3: Example of a browsing of a specific term

Other more advanced algorithms exist but are not described in this thesis, read [3, 24] for more details.

Chapter 5

The source-to-source debugger

This chapter represents my specific technical contribution performed during my internship in MISSION CRITICAL Australia Pty Ltd, It is mainly based on the article "The implementation technology of the Mercury debugger" [4].

Basically, the Mercury debugger 'mdb' generates low level C code for the debugging phase and it is sometimes hard to relate it with the original Mercury code. That is why MC asked me to create a source-to-source debugger (or **ssdb**) which transforms an input Mercury program into a new Mercury program, which allows its execution and its debugging at the same time. The advantage of this is that when the transformation are done at the Mercury level, the developer can then generate the code from any back-ends (both the C, the Java and the Erlang back-end) without major work needed.

In the following pages, I will present my experience. Firstly, this chapter introduces basic concepts of the new debugger. Secondly, it introduces the work done in the paper [4] by Zoltan Somogyi, this paper gives a theoretical structure for the transformation performed by the source-to-source debugger. Thirdly, the chapter explains some problems met during the implementation. After that, it shows what the generated code looks like, then it describes all commands available for the users, it proposes some improvements and finally, it shows the results of a performance test between the 'mdb' debugger and the new source-to-source debugger.

The source-to-source debugger is a trace-based debugger and is divided in two independent parts. The first part gets the code to debug and modifies it by adding code inside the body of each procedure. These lines, also called events, allow the debugger to give the runtime system control to the user. The second part of the debugger is used to navigate inside the code by jumping from an event to another one. The two parts of the ssdb will be called respectively the **generator** and the **navigator**.

5.1 Basic concepts

5.1.1 Events

The selection of event type (also call ports) is an important decision in traced-based debuggers, since such debuggers can only give control to the user at these events. Traditional Prolog debuggers have given control to the user at the four ports in Byrd's box model [3, 37]:

call A call event occurs just after a procedure has been called, and control has just reached the start of the body of the procedure.

exit An exit event occurs when a procedure call has succeeded, and control is about to return to its caller.

redo A redo event occurs when all computations to the right of a procedure call have failed, and control is about to return to this call to try to find alternative solutions.

fail A fail event occurs when a procedure call has run out of alternatives, and control is about to return to the rightmost computation to its left that has remaining alternatives which could lead to success.

These ports are clearly important in the understanding of the program behaviour because they describe the interaction of a procedure with its callers. The generator part of the debugger transforms a given source program in order to give information about each procedure and eventually to stop the execution of the navigator at these ports.

5.1.2 The stacks

The navigator has its two own stacks to manipulate the representation of each procedure. When the debugger enters in a procedure by its call port, some information about the procedure is put on one or both stacks. This information about a procedure is called a *frame*. Each frame has the following fields:

- The event number: each event has a unique event number. It represents at what step in the program the execution is.
- The call sequence number (CSN): a CSN is used to be able to detect which events are related together, each event within the same procedure has the same call sequence number.
- The depth: the depth gives the number of ancestors linking the current top call with the initial invocation of main. It is computed by the size of the stack. A call or a redo will increment the depth caused by a push on the stack. An exit or a fail will diminish the depth caused by a pop from the stack.

- The procedure ID: this structure contains the name of the procedure, and the name of the module which includes this procedure.

- The list of variables: this list contains all information about the procedure arguments. Each argument might be represented by one of the following constructions:

  - *unbound head_var(var_name, pos)* : this structure is used by an *output* argument in a call or redo port, at this moment, the argument does not have any value. At the exit or fail port, the argument become as *bound_head_var* argument.
  - *some [T] bound_head_var(var_name, pos, T)* : this structure is used by a bound argument, it is an *input* argument at any port or an output argument at exit or fail port.
  - *some [T] bound_other_var(var_name, T)* : in Mercury, some elements are sometimes neither *input* nor *output*, in this case, they are *unused*. As this structure is almost never used, read Mercury documentation for more details.

The *var_name* represents the name of the parameter, *pos* is the position in the list of arguments of the predicate, *T* is the value of the variable and finally, *some[T]* is a hidden structure which holds the type of the variable.

As said before, the ssdb has two stacks. One is the principal stack and it is modified at each event, the second one is used only to hold non-deterministic procedures information.

At each event of a procedure, the principal stack is modified as follows:

call	The navigator constructs a frame containing the information provided in the call port and pushes it on the stack at the beginning of the call.
exit	The frame is popped from the stack at the end of the exit.
redo	The same frame than the one pushed during the call event is popped on the stack at the beginning of the redo.
fail	The frame is popped from the stack at the end of the fail.

The second stack is used only to keep the non-deterministic procedures information. See *Too many operations are executed after a retry* at point 5.3.1. for more details.

call	The frame is push on the stack at the beginning of call.
exit	The frame is pop from the stack at the end of the exit.
redo	The stack is not modified.
fail	The stack is not modified.

5.1.3 Goal determinism

During the code transformation, the debugger has to detect what is the determinism of each goal. There are two possibilities:

- It is possible to ask to the compiler to detect it by itself. It is generally the best option because it avoids most of possible errors. The only drawback is that it consumes times to compute each determinism of each construction.
- The other option is to set the determinism category by hand. In this case, the programmer has to know all determinism rules. The determinism of each goal is inferred according to the following rules. These rules work with the two components of determinism category: whether the goal can fail without producing a solution, and the maximum number of solutions of the goal (0, 1, or more).

Here is how determinism has to be determined for each type of goal [12]:

Calls - p($arg_1, arg_2, ...arg_N$*)*

The determinism of a call comes from either the determinism declared or from the determinism from the called mode of the called procedure.

Unifications - Goal1 = Goal2

The determinism of unification is either det, semidet or failure, depending on its mode.
The assignment unification is deterministic. A unification that tests whether a variable has a given top function symbol is semideterministic, unless the compiler knows the top function symbol of that variable, in which case its determinism is det or failure. An equality test is always semideterministic.

Conjunctions - Goal1, Goal2

The conjunction succeeds if each conjunct succeed in the same execution. Otherwise, the conjunction will fail.

Disjunctions - Goal1; Goal2

A disjunction, that is not a switch, can fail if each disjunct fail. Otherwise, only one disjunct is required for the succeeding of the disjunction.

Switches

A disjunction is a switch if each disjunct has near its start a unification that test the same bound variable against a different function symbol.

Example 5.1.1 *switch example*

```
io.read_char(X, !IO),
(
    X = a,
    ...
;
    X = b,
    ...
;
    ...
)
```

A switch can fail if the various arms of the switch do not cover all the function symbols in the type of a switched-on variable. A switch can succeed several times if different arms succeed successively.

If-then-else - ***if*** *CondGoal* ***then*** *ThenGoal* ***else*** *ElseGoal*

If the *CondGoal* cannot fail, the *CondGoal* and *ThenGoal* form a conjunction and its determinism is computed accordingly. Otherwise, an if-then-else can fail if either the *ThenGoal* or the *ElseGoal* can fail. It fails if the *ElseGoal* fails and if at least one of the *CondGoal* and the *ThenGoal* fail. It can succeed more than once if any one of the *CondGoal*, the *ThenGoal* and the *ElseGoal* can succeed more than once.

Negations - \+ *Goal*

Negation goals succeed if the negated goal fails and vise-versa. The determinism of negations respects the following table:

Determinism of NOT G	Determinism of G
erroneous	erroneous
failure	det
det multi	failure
other	semidet

Table 5.1: Goal negation

5.2 Theoretical basis structure

5.2.1 The theoretical transformation

After having decided, in the precedent section, which events to be used, the next step was to choose how to arrange them to give control execution to the navigator after each event. As explained in [4], the simplest approach is to use a source-to-source transformation, as show by Ducasse and Noye.

Suppose the original predicate rule is as follow:

```
Head :- Body.
```

and we modify it by a source-to-source transformation. In other word, the debugger takes a source code in input and generates a new source code in output.

The source code in output looks like the following:

```
Head :-
   ( trace ( call , Head )
   ; trace ( fail , Head )
   ),
   Body,
   ( trace ( exit , Head )
   ; trace ( redo, Head )
   ).
```

The predicate *trace* introduces events around the *Body* of the original predicate rule. The *trace* predicate is deterministic and therefore, does not modify the determinism of the source code. The general idea behind the transformations is as follows. Where the program is executed under the supervision of the debugger, a call to the trace predicate allows the user to interact with the navigator.

However, the transformation above does introduce a number of disjunctions, which in the case of Mercury can change the determinism of the given procedure. This transformation breaks one of the invariants that the Mercury execution algorithm depends on, namely that a procedure declared to be det (i.e. to have exactly one solution) or semidet (i.e. to have either zero or one solution) will not leave behind any nondet stack frames (Mercury's equivalent of Prolog choice points) when it exits. Trying to get around this by simply saying that in debugging mode, all procedures are considered to be nondet or multi (and thus allowed to have more than one solution) would not work. In Mercury, I/O is only allowed in a deterministic context, so this change would in effect disallow I/O [4].

A simpler approach, which we adopted, is to vary the transformation depending on the determinism of the procedure. We transform deterministic procedure into:

```
Head :-
    trace ( call , Head ),
    Body,
    trace ( exit , Head ).
```

and semideterministic one to:

```
Head :-
    (
        trace ( call , Head ),
        Body
    ->
        trace ( exit , Head )
    ;
        trace ( fail , Head ), fail
    ).
```

For nondet and multi procedures (procedures that can succeed more than once) we use a variant of Ducass and Noy's transformation, which has the same operational semantics but leads to better code in Mercury, because the Mercury compiler generates more efficient code for nested disjunctions than for conjoined disjunctions [4]:

```
Head :-
    (
        trace ( call , Head ),
        Body,
        ( trace ( exit , Head )
        ; trace ( redo , Head ), fail
        )
    ;
        trace ( fail , Head ), fail
    ).
```

5.2.2 Practical Transformation

The theoretical model gives the skeleton for the code generation. This section follows the theoretical transformation schemas but now, we put more information to show how to allow the interactions between the user and the debugger.

The following procedure, which could represent any procedure in Mercury:

```
p( <arguments> ) :-
    <original body>
```

can be transformed, accordingly to its determinism, into one of the following possibility.

Deterministic model transformed (1 – 1 solution)

Contains a call event and an exit event.

```
1  p( <arguments> ) :-
2      promise < original purity > (
3          CallVarDesc = [ <arguments> ],
4          impure call_port ( ProcId, CallVarDesc ),
5          < original body >, % remaining outputs
6          ExitVarDesc = [ <arguments> ],
7          impure exit_port ( ProcId, ExitVarDesc, DoRetry ),
8          (
9              DoRetry = do_retry,
10             p ( <arguments> )
11         ;
12             DoRetry = do_not_retry,
13             % bind outputs
14         )
15     ).
```

This is how the theoretical deterministic schema is transformed into the practical schema. The *trace* predicate has been replaced in this case either by:

```
impure call_port ( ProcId, CallVarDesc )
```

or

```
impure exit_port ( ProcId, ExitVarDesc, DoRetry )
```

depending on the type of the event needed. These predicates transfer control to the navigator, providing it with the necessary information about the current procedure (*ProcId*) and the state of its arguments upon call (*CallVarDec*) and exit (*ExitVarDesc*).

The *impure* word is a purity concept used by Mercury and is not explained in details in this thesis as it is rarely used. Some predicates cannot be implemented within the paradigm of the logic programming, but it would be more convenient or more efficient to write such predicates with the facilities of Mercury. These predicates become then *impure* and the programmer needs to make it explicit for the compiler. Refer to *The Mercury Language Reference Manual* [12] on page 112 for more details on the purity.

The lines:

```
(
    DoRetry = do_retry ,
    p ( <arguments> )
;
    DoRetry = do_not_retry ,
    % bind outputs
)
```

is a construction which allows the *retry* command after the exit port. The *retry* command allows the debugger to go backwards in the execution of the program, it comes back to the call event which have the same CSN that the exit event retried.

Furthermore, to fully understand the code, take in note that:

CallVarDesc and *ExitVarDesc* are lists of *var_value*. Each *var_value* represents an argument of the predicate, properly encoded in the type expected by the debugger.

```
:- type var_value
    --->    unbound_head_var ( var_name, var_pos )           % output
    ;       some[T] bound_head_var ( var_name, var_pos, T )  % input
    ;       some[T] bound_other_var ( var_name, var_pos, T ) % unused
        .

:- type var_name == string.   % name of the argument

:- type var_pos == int.       % position of the argument in the <arguments>
```

Finally, note that this deterministic model transformed is also valid for the cc_multi determinism.

Semi-deterministic model transformed (0 – 1 solution)

The following transformation is performed by the debugger on every semi-deterministic predicates.

```
1  p ( <arguments> ) :-
2      promise < original purity > (
3          CallVarDesc = [ <arguments> ],
4          impure call_port ( ProcId, CallVarDesc ),
5          (
6              < original body> % remaining outputs
7          ->
8              ExitVarDesc = [ <arguments> ],
9              impure exit_port ( ProcId, ExitVarDesc, DoRetryA ),
10             (
11                 DoRetryA = do_retry,
12                 p ( <arguments> )
13             ;
14                 DoRetryA = do_not_retry,
15                 % bind outputs
16             )
17         ;
18             impure fail_port ( ProcId, CallVarDesc, DoRetryB ),
19             (
20                 DoRetryB = do_retry,
21                 p ( <arguments> )
22             ;
23                 DoRetryB = do_not_retry,
24                 fail
25             )
26         )
27     ).
```

This transformation is similar to the deterministic transformation except that there is a fail port, the body of the original predicate is placed in the condition of an if-then-else construction. This allows interactions with the navigator in a different way upon success or failure of the original body.

Please note that this transformation is also valid for the cc_nondet determinism.

Non-deterministic model transformed (0 – M solutions)

The transformation for the non-deterministic procedures is similar to others:

```
1  p ( <arguments> ) :-
2      promise < original purity > (
3          (
4              CallVarDesc = [ <arguments> ],
5              impure call_port ( ProcId, CallVarDesc ),
6              < original body>, % remaining outputs
7              ExitVarDesc = [ <arguments> ],
8              (
9                  impure exit_port ( ProcId, ExitVarDesc, DoRetryA ),
10                 (
11                     DoRetryA = do_retry,
12                     p ( <arguments> )
13                 ;
14                     DoRetryA = do_not_retry,
15                     % bind outputs
16                 )
17             ;
18                 impure redo_port ( ProcId, ExitVarDesc),
19                 fail
20             )
21         ;
22             impure fail_port ( ProcId, CallVarDesc, DoRetryB ),
23             (
24                 DoRetryB = do_retry,
25                 p ( <arguments> )
26             ;
27                 DoRetryB = do_not_retry,
28                 fail
29             )
30         )
31     ).
```

There is a disjunction on the call and fail port. If the execution goes into the call, there is a new disjunction between exit and redo port. Note that with a disjunction, more than one disjunct can succeed at a time. The redo port is used to try to find an alternative solution.

Please note that this transformation is also valid for multi determinism.

Failure model transformed (0 − 0 solution)

When the debugger reaches a failure goal, it is composed with a call event followed by a fail event. A retry is always possible but it will fail straight away.

```
1  p ( <arguments> ) :-
2      promise < original purity > (
3          CallVarDesc = [ <arguments> ],
4          impure call_port ( ProcId, CallVarDesc ),
5          < original body >,  % remaining outputs
6          FailVarDesc = [ ],
7          impure fail_port ( ProcId, FailVarDesc, DoRetry ),
8          (
9              DoRetry = do_retry,
10             p ( <arguments> )
11         ;
12             DoRetry = do_not_retry
13         )
14     ).
```

Erroneous model transformed (0 – 0 solution)

Finally, if the debugger reaches an erroneous goal, it should stop because otherwise the state of the debugger would become incoherent. In fact, as developed in the transformation, there is only a call port which pushes a frame on the stack, but this one will never be popped. Throwing an exception at this moment and propagate it to the caller could be added in future work.

So, if the debugger reaches an erroneous procedure, it will stop running. Hopefully, this kind of code is extremely rare and used in very particular context.

```
1 p ( <arguments> ) :-
2     promise < original purity > (
3         CallVarDesc = [ <arguments> ],
4         impure call_port ( ProcId, CallVarDesc ),
5         < original body >
6     ).
```

An erroneous procedure always stops the execution of the program. It happens, for example, when the execution enters in an infinite loop.

5.3 Practical structure

Obviously, an implementation of the above schemes is not straightforward because it requires advanced manipulations of the program variables (like the detection of the variable type). Some problems arose that makes that the implementation does not strictly follows the theory, mainly for the non-deterministic predicates. The explanation on encountered problems and the final detailed source-to-source transformations for non-deterministic predicates are shown in the following pages.

5.3.1 Non-deterministic model transformed –final release (0 – M solutions)

Two problems arose during the execution of non-deterministic procedures by the ssdb. Firstly, the debugger drops the event number and the call sequence number between an exit and a redo event. And secondly, the debugger executes too many operations if a retry occurs. Let us examine these problems in more detail.

First issue: The debugger loses the event number and the call sequence number

The problem with the non-deterministic code is that the procedure can be called more than once during the same execution. Remember the event sequence of a non-deterministic procedure: A non-deterministic procedure is represented by a call event, followed by a sequence of exit and redo events, and ended by a fail event.

At the call port, the debugger creates a frame containing all data of the current procedure: name of the procedure, name of the module in which the procedure is declared, the current event number of the call, the CSN (call sequence number), the depth, and the arguments of the procedure. This frame is then pushed on the stack.

At the exit port, the output arguments of the procedure are filled inside the frame stack. So the user can print them and performs some operations on them before the next call. Then, at the end of the exit event, the frame is popped from the stack, and all data concerning the event number and CSN (Call Sequence Number) during the call event are lost.

At the redo port, how to get the right CSN of the call event since it was popped from the stack? How to get the right event number if the user do a retry?

The first solution was to modify the generated code. If the CSN and event number are introduced and computed in the generated code, we avoid wrong number problems.

So, the generated code should look like the following:

```
1  p ( <arguments> ) :-
2      promise < original purity > (
3      (
4          CallVarDesc = [ <arguments> ],
5          get_current_csn(CSN),
6          get_current_event_number(EventNum),
7          get_csn_increment(CSNInc),
8          get_event_number_increment(EventNumInc1),
9          impure call_port ( ProcId, EventNumInc1, CSNInc, CallVarDesc ),
10         < original body>, % remaining outputs
11         ExitVarDesc = [ <arguments> ],
12         (
13             get_event_number_increment(EventNumInc2),
14             impure exit_port_non_det ( ProcId, EventNumInc2, CSNInc,
15                 ExitVarDesc),
16             %If Retry : go to fail_port_non_det
17         ;
18             get_event_number_increment(EventNumInc3),
19             impure redo_port ( ProcId, EventNumInc3, CSNInc, ExitVarDesc),
20             fail
21         )
22     ;
23         get_event_number_increment(EventNumInc4),
24         FailVarDesc = [ ],
25         impure fail_port_non_det ( ProcId, EventNumInc4, CSNInc,
26             CallVarDesc, DoRetryB ),
27         (
28             set_current_event_number(EventNum),
29             set_current_csn(CSN),
30             DoRetryB = do_retry,
31             p ( <arguments> )
32         ;
33             DoRetryB = do_not_retry,
34             fail
35         )
36     )
37 ).
```

This solution is easy to implement but very greedy in time resources during the code generation. Each new predicate inserted (*get_current_csn, get_current_event_number, etc.*) consumes a lot of time to be inserted in the original source code.

There is a way to significantly improve the code generation time. It is possible to work without "getters" and "setters" of the Event Number and the CSN. As we lose the Event Number and the CSN when the frame is popped at the exit event, another solution is to keep this frame somewhere. So the simplest way is to implement a second stack, which will contain only the multi/non-deterministic procedure frame.

As a reminder of operations on multi/non-deterministic procedures, the order of the events in such procedures is the following:

- For a nondet procedure : there is a call event, followed by *zero* to n exit and redo events and finally by a fail event.
- For a multi procedure : there is a call event, followed by *one* to n exit and redo event and finally by a fail event.

The difference is not at the call or fail port level as the frame is simply pushed at the call event and popped at the fail event on both stacks. This means of course that the difference is between the call and fail port. When an exit event occurs, only the principal stack is popped and the procedure frame from the call port remains on the nondet stack. When the redo event occurs, the frame on the nondet stack is called back and pushed on the principal stack. To call back the frame, the debugger just needs the name of the procedure, the name of the module in which this procedure is and the depth of the call. In this way, a recursive call is handled as any other procedure calls.

Second issue: Too many operations are executed after a retry

With the non-deterministic procedure, when a retry happens, the remaining branches of the decision tree are not destroyed. In other words, when a retry is executed in the middle of a multi/nondet procedure, the execution stops at the current call point and creates a new decision point in the SLD tree and calls the new procedure. When this call is over, the debugger comes back to the old decision point of the caller and terminates its execution. If we have the following example:

Example 5.3.1 *Too many operations executed*

```
:- pred main(io::di, io::uo) is cc_multi.
main(!IO) :-
    unsorted_solutions(p ,Solutions).

:- pred p(list(int)::out) is multi.
p(A) :-
    (
        append([1], [2], [])
    ->
        A = []
    ;
        append([2], [4], Xs),
        append(A, _, Xs)
    ).
```

The solutions of this little program are the following: *([2], [4], [2,4])*

If we examine the execution of this program with the ssdb navigator, we could get an interaction like the following:

Example 5.3.2 *Too many operations executed*

Event #	Call #	Depth	Event Type	Procedure	Result
1	1	1	call	main	
2	2	2	call	p	
3	2	2	exit	p	[ ]
4	2	2	redo	p	
5	2	2	exit	p	[2]
—>retry : new decision point					
2	2	2	call	p	
3	2	2	exit	p	[ ]
4	2	2	redo	p	
5	2	2	exit	p	[2]
6	2	2	redo	p	
7	2	2	exit	p	[4]
8	2	2	redo	p	
9	2	2	exit	p	[2,4]
10	2	2	redo	p	
11	2	2	fail	p	
—>end of retry : finish of the old decision point					
12	2	2	redo	p	
13	2	2	exit	p	[4]
14	2	2	redo	p	
15	2	2	exit	p	[2,4]
16	2	2	redo	p	
17	2	2	fail	p	
18	1	1	exit	main	

Table 5.2: Wrong event number in the debugger

In this case, the solutions are computed more than one time and the event number becomes wrong when the "retried" procedure is finished. The correct event number at the end of the execution should be 12, and not 18. This is caused because the compiler executes both decisions in the tree instead of only the decision created by the retry. The solution was to slightly modify the non-deterministic model.

```
1  p ( <arguments> ) :-
2      promise < original purity > (
3          (
4              CallVarDesc = [ <arguments> ],
5              impure call_port_nondet ( ProcId, CallVarDesc ),
6              < original body>, % remaining outputs
7              ExitVarDesc = [ <arguments> ],
8              (
9                  impure exit_port_nondet ( ProcId, ExitVarDesc),
10                 %If Retry : go to fail_port_non_det
11             ;
12                 impure redo_port_nondet ( ProcId, ExitVarDesc),
13                 fail
14             )
15         ;
16             FailVarDesc = [ ],
17             impure fail_port_nondet ( ProcId, CallVarDesc, DoRetryB ),
18             (
19                 DoRetryB = do_retry,
20                 p ( <arguments> )
21             ;
22                 DoRetryB = do_not_retry,
23                 fail
24             )
25         )
26     ).
```

The code, generated for the retry just after the exit port, has been removed. All ports have been renamed as well. In theory, when the user does a retry after an exit port of non-deterministic procedures, the program jumps at the matching fail port of this procedure. All operations between these two points have to be ignored, then the retry is automatically called when the program reaches the fail port. In this way, we keep the right event number. Such a jump in the code could be annoying if several operations take a while to execute themselves (I.e. if the program needs to access a lot to the hard disk). Otherwise, even several million operations are almost instantaneous.

Nevertheless, there is a limitation. As the generated code is executed by the compiler, the operations are computed. Considering that the runtime system is not giving back the control to the user, it remains invisible for him. But, if the user prints the result set with a predicate such as *unsorted_solutions/2*, same solutions appear more than once. The debugger is going to live with this constraint.

5.4 The generated code

If we take back the example program in annexe 2, we can isolate the deterministic predicate 'p', the code for this predicate is:

```
:- pred p(int::in, int::out) is det.

p(X, Y) :-
    g(X, Y).
```

The Mercury code generated by the debugger is shown on the next page.

The low-level C code generated by the Mercury debugger is closer to the assembler than a classic C code. As said before, the source-to-source debugger can use any of the back-ends that Mercury has (low and high-level C code, Java, .NET and Erland back-end) making it easier to be understood by a human and to do the correspondence with the Mercury code.

After the generator of the ssdb has generated the code, the navigator allows the user to navigate through it with different commands. Read the next section '**5.5. The commands of the ssdb**' to know what commands are available with the navigator.

The Mercury code for the predicate 'p' is the following:

```
1  :- mode p((builtin.in), (builtin.out)) is det.
2  test.p(X, Y) :-
3    promise_pure (
4      ModuleName = "test", % Name of the module
5      PredName = "p",      % Name of the procedure
6          % ProcId construction
7      ProcId = ssdb.ssdb_proc_id(ModuleName, PredName),
8      EmptyVarList = list.[], % Creation of the parameter empty list
9      VarName = "Y",          % Name of the argument
10     VarPos = 1,             % Pos of the argument
11     VarDesc = ssdb.unbound_head_var(VarName, VarPos), % Argument type
12     FullListVar = list.[VarDesc | EmptyVarList], % Add to the list
13     VarName = "X",
14     VarPos = 0,          % Ditto with the second argument
15         % Constructor (because it is an input argument)
16     TypeCtorInfo_15 = type_ctor_info("", "int", 0),
17     VarDesc = ssdb.bound_head_var(TypeCtorInfo_15, VarName, VarPos, X),
18     FullListVar = list.[VarDesc | FullListVar],
19     impure ssdb.handle_event_call(ProcId, FullListVar), % Call event
20     test.g(X, V_18),        % Body of the procedure
21     EmptyVarList = list.[], % Creation of the second list
22     VarName = "Y",          %
23     VarPos = 1,             % Only modified argument are generated again
24     TypeCtorInfo_22 = type_ctor_info("", "int", 0),
25     VarDesc = ssdb.bound_head_var(TypeCtorInfo_22, VarName, VarPos, V_18),
26     FullListVar = list.[VarDesc | EmptyVarList],
27     FullListVar = list.[VarDesc | FullListVar],
28         % Exit event
29     impure ssdb.handle_event_exit(ProcId, FullListVar, DoRetry),
30     ( % cannot_fail switch on 'DoRetry'
31       % DoRetry has functor ssdb.do_retry/0
32       test.p(X, Y)          % Switch for the retry
33     ;
34       % DoRetry has functor ssdb.do_not_retry/0
35       Y = V_18
36     )
37   ).
```

The lines 3 to 19 and 21 to 37 have been generated by the ssdb. This shows the efforts that the ssdb have to do even for a very small predicate.

5.5 The commands of the ssdb

The debugger generator uses five different types of commands:

- *forward* commands allow the user to go forward from a step to another by jumping.
- *backward* command (composed only with the retry command) sets the debugger to a previous state.
- *browsing* commands allow the user to see parameters or stack and their features at each step.
- *breakpoint* commands enable the user to set breakpoint and manage them like he wants.
- *other* commands like the exit or help command.

The current source-to-source debugger does not handle yet all commands that the 'mdb' does. But the more important and useful ones are present and can be used.

In the following pages, note that an ancestor is a call before the current call.

5.5.1 Forward movement commands

`'step' or 's'`

Forwards to the next event.
An empty command line is interpreted in the same way.

`'next [num]' or 'n [num]'`

Continues execution until the program reaches the next *num*'th ancestor of the call to which the current event refers. The default value of *num* is one, in this case, the navigator skip all events until it reaches the next event with the same CSN. For example, if *num* is 2, it goes to the next event of the parent of the current call. The debugger reports an error if the execution is already at the end of the specified call.

`'finish [num]' or 'f [num]'`

Continues execution until the program reaches a final (exit or fail) port of the *num*'th ancestor of the call to which the current event refers. The default value of *num* is one, which means skipping to the end of the current call. For example, if *num* is 2, it finishes (go to the last event of the procedure) the call of the parent of the current procedure. The debugger reports an error if execution is already at the desired port.

`'goto num' or 'g num'`

Continues execution until the program reaches the event number *num*.

`'continue' or 'c'`

Continues execution until it reaches the end of the program or a breakpoint enabled. Read the '**Breakpoint commands**' subsection for more details on breakpoint commands.

5.5.2 Backward movement command

`'retry [num]' or 'r [num]'`

If the optional number is not given, restarts execution at the call port of the call corresponding to the current event. If the optional number is given, restarts execution at the call port of the call corresponding to the *num*'th ancestor of the call to which the current event belongs. For example, if *num* is 2, it restarts the parent of the current call.
A retry over I/O actions are not safe. A note concerning this problem is explained in *5.6 Potential expected developments* section.

5.5.3 Browsing commands

`'print' or 'p'`

Prints the values of all the known variables in the current environment.

`'stack'`

Prints each procedure on the stack with an ordinal number and their respective arguments.

`'browse name'`

Invokes the interactive term browser to browse the value of the variable in the current environment with the given name.

The interactive term browser allows one to examine particular subterms. The depth and size of printed terms may be controlled. The displayed terms may also be clipped to fit within a single screen.

`'up'`

Set the current environment to the stack frame of the num'th level ancestor of the current environment (the intermediate caller is the first-level ancestor).

This command will report an error if the current environment does not have the required number of ancestors, or if there is an execution trace information about the requested ancestor, or if there is no stack trace information about any of the ancestors between the current environment and the requested ancestor.

`'down'`

Set the current environment to the stack frame of the num'th level descendant of the current environment (the procedure called by the current environment is the first-level descendant).

This command will report an error if there is no execution trace information bout the requested descendant.

5.5.4 Breakpoint commands

`'break module_name procedure_name' or 'b module_name procedure_name'`

Puts a break point on the specified procedure. By default, the break point is enabled.

If a procedure is overloaded (e.g. : *bar.baz/3* and *bar.baz/4* or *bar.baz/3* with another mode), the execution will stop at each occurrence of the procedure overloaded.

If a procedure has more than one module_name, like: *foo.bar.baz.procedure_name*. Then, all module names should be put in the module_name field.

`'break info'`

Lists details, status and print lists of all break points.

`'disable num'`

Disables the break point with the given number.

`'disable *'`

Disables all break points.

`'enable num'`

Enables the break point with the given number.

`'enable *'`

Enables all break points.

`'delete num'`

Deletes the break point with the given number.

`'delete *'`

Deletes all break points.

5.5.5 Miscellaneous

`'help'`

Prints summary information about all the available commands.

`'exit'`

Quits the debugger and abort the execution of the program.

End-of-file on the debugger's input is considered as an exit command.

5.6 Potential expected developments

This section proposes some improvements that could greatly improve the user friendliness and the efficiency of the debugger:

- Add condition, then, else, disjunction and switch port [4].
- Handle exception.
- IO tabling (see User Guide [3] p25).
- Tab completion.
- Command history.
- Improve time performance during the code generation and execution.

5.6.1 Addition of other ports

In addition to the four interface (or external) ports already implemented, a fifth one should be added to handle exceptions:

excp A excp event occurs when control leaves a procedure and returns to its caller due to an uncaught exception.

In addition, currently, the debugger does not provide any information about the internal execution of the procedure. We should create event types corresponding to every kind of decision about the flow of control. These internal event types are the following: *if part, then part, else part, conjunction, disjunction, switch and negation, negf, negs* [4].

The counterpart of these added events will be a significant time consuming code generation. As the code generation is very slow, an option should be used to set on/off the generation of these internal events.

5.6.2 Handle exceptions

Each time that the program reaches a point where an exception is thrown, the execution of the debugger should stop and show the state at this point without crashing. Note that the exceptions discussed here are similar to the exceptions in some imperative languages, such as Java. To handle exceptions, the debugger has to manage built-in procedures.

In Mercury, the special predicate *throw/1* is used to throw exceptions. Calling *throw/1* causes its arguments to be thrown as an exception.

Example 5.6.1 *Utilisation of the throw/1 predicate*

```
divide(N, D, Q) :-
    ( if D = 0 then
        throw(division by zero)
    else
        Q = N / D
    ).
```

If a thrown exception is not caught, then the program aborts. For procedures which can succeed at most once, a *try* predicate can be used to handle it. For example:

Example 5.6.2 *Utilisation of the try predicate*

```
maybe_divide(D, N, MaybeQ) :-
    try(divide(D, N), Result),
    ( if Result = succeeded(Q) then
        MaybeQ = yes(Q)
    else
        MaybeQ = no
    ).
```

In this example, the first argument to try is the closure which may throw an exception. The predicate *divide/3* is currified to send only an output argument. Once the procedure has been executed, the second argument is unified with one of the three following possibilities:

- *succeeded(R)*, if the call succeeds, where R is bound to the output of the procedure,
- *failed*, if the call failed or
- *exception(E)*, if the call throws an exception, E is bound to the exception.

The *try_all(Goal, MaybeException, Solutions)* functor is used for procedures that may succeed more than once. If no exception is thrown by *Goal*, than *MaybeException* is bound to '*no*' and *Solution* is bound to the list of solutions found. Otherwise, if *Goal* throws an exception, *MaybeException* is bound to '*yes(E)*' and *Solutions* to the list of solutions found before the exception. When an exception is thrown, a *excp* event is generated and have the same CSN than the matching *call* event in the stack.

5.6.3 I/O tabling

As already explained, it is a well-known fact that we can not do a retry through an IO operation.

For example, if a procedure reads a file character by character, during the first execution, the procedure will read and return the first character, then, if we do a retry, the program should send the first character again and not the second one. To achieve this, we need to remember what IO was done before the retry and send the result of each operation when we do a retry.

We will illustrate the problem by means of a small example which uses IO operations:

Example 5.6.3

```
1  :- import_module io.
2  :- import_module require.
3
4  :- pred main(io::di, io::uo) is det.
5
6  :- implementation.
7
8  main(!IO) :-
9      io.open_input("../bar/baz.txt", IOResult, !IO),
10     (
11         IOResult = io.error(ErrInput),
12         error(io.error_message(ErrInput))
13     ;
14         IOResult = ok(Stream),
15         io.read_file_as_string(Stream, MaybeFileAsString, !IO),
16         (
17             MaybeFileAsString = ok(FileAsString),
18             io.write_string(FileAsString, !IO)
19         ;
20             MaybeFileAsString = error(_, ErrReadFileAsString),
21             error(io.error_message(ErrReadFileAsString))
22         ),
23         io.close_input(Stream, !IO)
24     ).
```

In this very small program, there are some IO operations, if we do a retry on:

io.open_input	Which opens a file to perform operations on it. We should not open twice the same file for an evident reason of performance: If we have either a huge file or thousands of files, it will quickly overheads the memory.
io.read_file_as_string	Which reads a string in the opened file. We have to send back the string read by the reading operation.
io.write_string	Which writes a string. We have to write the same string as the first time.
io.close_input	Which closes a file, a file already closed throws an exception.

The paper on *Idempotent I/O for safe time travel* [35] explains clearly all problems met and propose some solutions.

5.6.4 Tab completion

Command line completion is a common feature of command line interpreters, in which the program automatically fills in partially typed tokens. Depending on the specific interpreter and its configuration, these elements may include commands, arguments, environment variable names and other entities. Command line completion is often invoked, by default, by pressing the tab key and frequently called tab completion.

To use this feature, a table must be kept in memory and be filled in at the start of the debugger. The table needs to hold all procedure and function names of the current debugged program. Implementing this feature must (partially) be done in C because Mercury does not handle keystroke function.

5.6.5 Command history

Having the command history available is a common feature in a program that interacts with the user through a command line interface. Command history involves making previously-entered commands (usually up to some limit) easily available to enter once again at the command line. The usual method is for the user to use the Up (and Down) keyboard arrow keys to navigate through the command history, but some programs also offer the facility for the user to press a certain function key which will show a menu of recent commands, from which the user can select one by typing a number.

Command history takes advantage of the fact that the user may want to execute the same command many times, such as a developer frequently compiling and running a program, or the new command may be a small modification of a previous one, hence requiring little typing to modify it. It therefore saves a lot of typing for the user and increases the speed and accuracy of input to the computer.

5.6.6 Manage built–in procedures

When the code is generated, the option "–no-ssdb" is used to avoid handling built-in procedure by ssdb. In fact, the built-in procedure is implemented with other built-in procedures and an infinite loop may occur during the code transformation. To avoid this, an independent variable is used to disable the source-to-source debugger in the second level of built-in procedures. Be aware that this improvement could greatly reduce the speed of the debugger, in that way, an option should be created to enable or disable this feature.

5.6.7 Improve time performances

The performances of the debugger could be greatly improved by some code modifications.

- The ssdb should generate only one list of variables instead of two during the list argument generation. Currently, the debugger generates a new argument list during the call and the exit of the procedure. A unique list should be used. Arguments are added to the list during the call. But during the exit event, only output arguments have to be added to the list. Some modifications in the 'print' command are necessary to compute the list in order to only print the relevant arguments.

- Limit the number of appended lists in the implementation.

- Find a way to enable the optimisations. Most of the optimizations for the debugger are turn off for now. Maybe some modifications in the code could enable more optimizations to boost the debugger speed.

5.6.8 Others

Some other exotic features would improve the user friendliness of the debugger:

- The possibility to add options in command line. 'mdb' enables the user to add options in several commands like print output in a different way, ask more details, etc.

- A parameter file could be used to set some parameters for the debugger, during the second start up of the debugger, it will simply copy these parameters to use

them. These parameters could modify line length, number of lines on the screen, the formatting with 'pretty_printer' or with another similar library.

- Many other commands exist and might be added, such as the call to the declarative debugger for example.

- The ssdb does not handle the multi-threading (or parallelism).

5.7 Performance Test Results

The Mercury compiler is a huge and complex program with more than a hundred thousand lines of code. It is mainly a combination of three different languages: Mercury, Nu-Prolog and SISCtus Prolog. Basically, when a new feature is added to the compiler, one has to rebuild it by using the *bootcheck* command which demonstrates that the added feature does not cause any unexpected modifications in an other part of the debugger.

To complete this overview about the code transformation implemented as an additional phase within the Melbourne Mercury Compiler, we will show the influence of the code transformation during the execution time. To perform this task, we will use it to check the six most important classes of the compiler, important in term of study subjects (number of lines of code, complexity, etc.). The Table 5.3 shows the six modules used for the performance test. It also gives an idea in term of lines of code (LOC) of these modules.

Module name	LOC
typecheck.m	2939
polymorphism.m	3371
code_info.m	4410
mercury_compile.m	5275
llds_out.m	5676
modules.m	8454
TOTAL	30125

Table 5.3: The six most interesting modules used for the performance test

As explained before, Mercury uses distinct grades to compile its applications. Each of them uses specific optimizations and therefore, takes different times to compile.

The *asm_fast.gc* is the most common grade used as it contains all optimizations for Mercury.

The *asm_fast.ssdebug.gc* introduces the transformations from the source–to–source debugger but unfortunately turn off some optimizations for implementation purpose.

The next table show the slowdown due to the transformations performed with the *asm_fast.ssdebug.grade* during the execution time.

Grade	Compilation time (in sec.)
asm_fast.gc	17.03
asm_fast.ssdebug.gc	121.93

Table 5.4: Mercury's grade speed slowdown

The ratio is 7.15 times slower with ssdb enabled. This ratio is due, on one hand, by all the optimizations turned off, and on the other hand, by the difference of level that code transformations operate on (The higher the transformations operate, the slower it will be).

Note that this test was done on:

- Intel Core 2 Duo CPU T7300 ©2.00GHz
- Ubuntu
- Without Internet connexion and as few processes running as possible
- Without any extra flags

Chapter 6

Conclusion

Depending of the programming language used and therefore the available debuggers, the difficulty of software debugging greatly varies.

Logic languages have demonstrated their effectiveness to implement powerful and efficient debuggers with less effort compared to classic imperative languages. High-level programming languages, such Java, make debugging easier because they have features such as exception handling that make real sources of devious behaviour easier to spot. In C and other lower-level programming languages, bugs are often difficult to identify and finding the root cause of the problem may take hours.

Traditionally, most people have equated logic programming with Prolog, and have concluded that logic programming is not suitable for writing application programs except in narrow domains [12]. But MC has proved the opposite. They use an approach based on ontologies to formally describe the problem domain. This approach has enabled MC to successfully develop highly complex operational applications within a defined budget and timeframe. This approach is known as ODASE, Ontology Driven Architecture for Software Engineering.

The first complex application coded in Mercury was for FOREM, the regional unemployment agency of Wallonia in Belgium (with 3000 employees and an annual budget of 250 million euros). FOREM is continually facing complex and changing regulations which directly impact many of its business processes. In the past, several contractors had failed to develop a satisfactory system able to support a new employment programme. FOREM asked MC to develop such a system with the following requirements: 3-tier architecture with clear separations, Internet-ready with strong performance and robust security, furthermore it should be adaptative in the sense that it easily allows the business process modifications necessary to cope with a continuously changing regulation. Finally, MC produced a "light" client of 22,000 lines of Java code, although only dealing with presentation layout, whereas the application server requires only 18,000 lines of Mercury code. This bodes well for Mercury's performance, cost (both development and maintenance) and re-

liability [38, 39].

Mercury, with its strong types, mode and determinism system and all its other embedded features allow a high percentage of common errors to be identified during compile-time. Therefore, coding in Mercury, can be substantially faster and more cost effective than in any other languages as illustrated by the FOREM case. That system has proven to be suitable to create debuggers. Three different debuggers had been created in the past: a simple procedural debugger similar to the tracing system of Prolog implementations, a prototype declarative debugger, and a debugger based on the idea of automatic trace analysis [4], the new complementary debugger has been implemented, in a similar way than existing procedural debuggers, which might be used with any back-ends proposed by Mercury. Now, the possibility to generate a different code than the low-level C code, such as Erlang or high-level C code, makes easier to match the relation with the Mercury code. However, the current implementation of the source-to-source debugger needs various improvements.

Another part of my work was take into account all the remarks proposed to improve the speed of the new debugger. The point was important as stated by Zoltan Somogyi at the beginning of my task: *"There is no quick execution, you just have the choice between a slow or a very slow execution time depending on the implementation"*.

In the current state, the development of the source-to-source debugger is not over. Handling exceptions should be the next functionality to be added. The debugger should be able to handle exceptions, go through it without crashing and show the state of the program before the exception. A second very useful future work should be the debugging of built-in procedures, to go through the program step by step in every procedure and be able to see whether a variable takes a value that is not expected by the programmer. Other improvements are necessary to keep the program consistent during the debugging such as I/O tabling. New features should also be added to the navigator as the Tab completion and the Command History, two common features in debuggers. Finally, some speed improvements are certainly possible if we reduce the number of list appends and if we limit the generation of only one list of argument for each procedure. However, the main commands for the new debugger are available and it handles perfectly all determinisms; this was the main objectives of my work.

Bibliography

[1] Inquiry board. ariane 5, flight 501 failure report. `http://www.cs.unibo.it/laneve/papers/ariane5rep.html`, July 1996. Computing Science Departement, Glasgow University.

[2] Mark Brown and Zoltan Somogyi. Annotated event traces for declarative debugging. `http://www.cs.mu.oz.au/mercury`, 2005. University of Melbourne.

[3] Fergus Henderson and al. *The Mercury User's Guide*. University of Melbourne, Melbourne, Australia, 2006.

[4] Zoltan Somogyi and Fergus Henderson. The implementation technology of the mercury debugger. *Electronic Notes in Theoretical Computer Science*, 30(4), 1999.

[5] Michel Vanden Bossche-Marquette. Le projet phenix. pourquoi les échecs informatiques ne sont pas l'exception, Mars 2007. Mission Critical.

[6] G. Comyn. Programming in 2010? a scientific and industrial challenge. In J. W. Lloyd, editor, *Computational Logic: Symposium Proc.*, pages 202–203. Springer, Berlin, Heidelberg, 1990.

[7] Tom Gruber. What is an ontology? `http://www-ksl.stanford.edu/kst/what-is-an-ontology.html`, 1990. European Computer-Industry Research Centre (ECRC).

[8] Michel Vanden Bossche-Marquette. Odase references, 1995-2008. Mission Critical.

[9] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1-3):17–64, 1996.

[10] Zoltan Somogyi and Fergus Henderson. Mission critical it site. `http://www.missioncriticalit.com/`, November 2006.

[11] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1984, 1987. Second, Extended Edition.

[12] Fergus Henderson and al. *The Mercury Language Reference Manual*. University of Melbourne, Melbourne, Australia, 2006.

[13] Wim Vanhoof. Programmation fonctionnelle et logique, Academic year 2004-2005. Course at the University of Namur.

[14] Jean-Marie Jacquet. Technique d'intelligence artificielle, Academic year 2004-2005. Course at the University of Namur.

[15] Zoltan Somogyi and Fergus Henderson. The mercury project on melbourne university. `http://www.cs.mu.oz.au/research/mercury/`. University of Melbourne.

[16] Ralph Becket. *Mercury tutorial*. University of Melbourne, Melbourne, Australia, 2006.

[17] F. Henderson, Z. Somogyi, and T. Conway. Determinism analysis in the mercury compiler, 1996.

[18] J. Launchbury. Lazy imperative programming. In *Proceedings of the ACM SIGPLAN Workshop on State in Programming Languages, Copenhagen, DK, SIPL '92*, pages 46–56, 1993.

[19] Don Sannella. Lazy and eager evaluation. `http://homepages.inf.ed.ac.uk/dts/fps/lecture-notes/lazy.pdf`, October 2005. Lecture Note 1, Course at the University of Edinburgh.

[20] S. Taylor. Optimization of mercury programs, 1998.

[21] Wikipedia. `http://en.wikipedia.org/wiki/`.

[22] Ian MacLarty, Zoltan Somogyi, and Mark Brown. Divide-and-query and subterm dependency tracking in the mercury declarative debugger. In Clinton Jeffery, Jong-Deok Choi, and Raimondas Lencevicius, editors, *Proceedings of the Sixth International Workshop on Automated Debugging*, pages 59–68. ACM, 2005.

[23] Ian MacLarty and Zoltan Somogyi. Controlling search space materialization in a practical declarative debugger. In Pascal Van Hentenryck, editor, *Practical Aspects of Declarative Languages, 8th International Symposium*, volume 3819 of *Lecture Notes in Computer Science*, pages 31–44. Springer, 2006.

[24] Ian MacLarty. *Practical Declarative Debugging of Mercury Programs*. PhD thesis, PhD thesis, Department of Computer Science and Software Engineering, University of Melbourne, Melbourne, Australia, July 2005.

[25] Paul Brna May. Programming a first course.

[26] David R. Hanson and Mukund Raghavachari. A machine-independent debugger. *Software - Practice and Experience*, 26(11):1277–1299, 1996.

[27] Debugging "c" and "c++" programs using "gdb". `http://mia.ece.uic.edu/~papers/WWW/debugging/debugging-with-gdb.html`.

[28] Richard Stallman and al. *Debugging with GDB The GNU Source-Level Debugger*. Melbourne, Australia, February 2004. Ninth Edition.

[29] John Gimore. *GDB Internals A guide to the internals of the GNU debugger*, February 2004. Cygnus Solution Revision. Second Edition.

[30] Laura Bennett. Java debugging. `http://ibm.com/developerWorks`.

[31] Sun microsystem. jdb the java debugger. `http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/jdb.html`.

[32] Mireille Ducassé and Erwan Jahier. *A High-level Debugging Environment for Prolog OPIUM 3.1 User Manual*. Munich, Germany, 1991. European Computer-Industry Research Centre GmbH.

[33] Mireille Ducassé and Erwan Jahier. *An Automated Debugger for Mercury - Opium-M 0.1 User and Reference Manuals*. Rocquencourt, France, May 1999. Institut National de Recherche en Informatique et en Automatique (INRIA).

[34] Mireille Ducassé and Erwan Jahier. *An automated debugger for Mercury Morphine 0.1 User and Reference Manuals*. Rennes, France, May 1999. Institut de Recherche en Informatique et Système Aléatoire (IRISA).

[35] Zoltan Somogyi. Idempotent i/o for safe time travel, 2003.

[36] Chris Speirs, Zoltan Somogyi, and Harald Sondergaard. Termination analysis for mercury. In *Static Analysis Symposium*, pages 160–171, 1997.

[37] Paul Brna. The box model of execution. `http://computing.unn.ac.uk/staff/CGPB4/prologbook/node47.html`, May 1999.

[38] Second Chance Kung-Kiu. Logic programming for software engineering:, 2002. University of Manchester.

[39] Michel Vanden Bossche. High-quality and predictable mission-critical software. `http://www.missioncriticalit.com/white/papers.htm`, 2001. Mission Critical IT.

[40] Zoltan Somogyi and Fergus Henderson. *The Mercury Library Reference Manual*. Melbourne, Australia, 1995-1997, 1999-2005. University of Melbourne.

[41] Ali E. Abdallah. Reduction strategies and lazy evaluation. `http://myweb.lsbu.ac.uk/~abdallae/units/fp/reduce.pdf`. Course at London South Bank University.

[42] J. Launchbury. Lazy imperative programming. In *In Proceedings of the ACM SIGPLAN Workshop on State in Programming Languages*, pages 46–56, Copenhagen, DK, 1993. SIPL92.

[43] Z. Somogyi, F. Henderson, T. Conway, and R. O'Keefe. Logic programming for the real world, 1995.

[44] An introduction to the mercury source code and tools. `http://www.mercury.csse.unimelb.edu.au/information/developers/developer_intro.html`. University of Melbourne.

Annexes

Annexe 1

How to use the Mercury compiler

In order to compile a program, the Mercury compiler has to be invoked with '*mmc* − −*make* < *filename* >'.

Example 6.0.1 *file find_path.m compiled:*

```
$ mmc -- make find_path
$ ./find_path
```

Which gives the following result:

Example 6.0.2 *Result of find_path.m once executed:*

```
[arc(a,c), arc(c,d)]
```

And it creates an executable '*filename*'. For programs that consist of more than one source file, it is the command Mmake which ensures that steps are done in the right order and not repeated.

The Mercury compiler uses grades which set a brunch of options. These options are related to the target language, the garbage collection strategy, the profiling technique, the debugging, the parallel generation of code and many other parameters. The entire program must be compiled with the same settings of these options, and it must be linked to a version of the Mercury library which has been compiled with the same settings [35]. This is especially important for large programs written in Mercury like the Mercury compiler itself.

Example 6.0.3 *find_path.m from* ***Example 3.1.3****, compiled with a grade.*

```
$ mmc  debug.asm_fast.gc my_member
```

In this example, the program is compiled with the *debug* grade, which allows the programmer to use the Mercury debugger on this program. The *asm_fast* grade uses different feature to speed up the compilation and execution time of the applications. And finally, the *.gc* grade called the Hans Boehm's conservative garbage collector for C (see the Mercury User Guide [3] for all details).

Annexe 2

```
1  % Example of a deterministic code.
2  %
3  :- module test.
4  :- interface.
5
6  :- import_module io.
7
8  :- pred main(io::di, io::uo) is det.
9  :- pred p(int::in, int::out) is det.
10 :- pred g(int::in, int::out) is det.
11 :- implementation.
12
13 main(!IO) :-
14     p(1, X),
15     io.write(X, !IO),
16     io.nl(!IO).
17
18 p(X, Y) :-
19     g(X, Y).
20
21 g(X, X).
```

```
1  %Example of a semi-deterministic code.
2  %
3  :- import_module io.
4  :- import_module list.
5
6  :- pred main(io::di, io::uo) is det.
7  :- pred p(int::in, int::out) is semidet.
8  :- pred g(int::in, int::out) is det.
9  :- implementation.
10
11 main(!IO) :-
12      (
13          p(1, X)
14      ->
15          io.write(X, !IO),
16          io.nl(!IO)
17      ;
18          io.write_string("\np failed", !IO)
19      ).
20
21 p(X, Y) :-
22      g(X, Y),
23      L = [2,3,4],
24      list.delete(L, X, _L0).      % list.delete/3 is semidet code.
25
26 g(X, X).
```

```
1  % Example of a non-deterministic code.
2  %
3  :- import_module io.
4  :- import_module list.
5  :- import_module solutions.
6
7  :- pred main(io::di, io::uo) is cc_multi.
8  :- pred p(list(int)::out) is multi.
9  :- pred g(list(int)::out) is multi.
10 :- implementation.
11
12 main(!IO) :-
13     unsorted_solutions(p ,Solutions),
14     io.write(Solutions, !IO),
15     io.nl(!IO).
16
17 p(A) :-
18     promise_pure
19     (
20         g(A)
21     ).
22
23 g(A) :-
24     (
25         append([1], [2], [])
26     ->
27         A = []
28     ;
29         append([2], [4], Xs),
30
31         append(A, _, Xs)          % this is nondet code.
32     ).
```

```
1  % Example of a failure code.
2  %
3  :- import_module io.
4
5  :- pred main(io::di, io::uo) is det.
6  :- pred p(int::out) is det.
7  :- pred g(int::out) is failure.
8  :- implementation.
9
10 main(!IO) :-
11     (if
12         p(A),
13         g(A)
14     then
15         io.write("A = ", !IO),
16         io.write(A, !IO),
17         io.nl(!IO)
18     else
19         io.write_string("failed", !IO)
20     ).
21
22 p(A) :-
23     A = 10.
24
25 g(_A) :-
26     fail.           % this is failure code.
```

```
1  % Example of an erroneous code.
2  %
3  :- import_module io.
4
5  :- pred main(io::di, io::uo) is det.
6  :- pred loop(int::in) is erroneous.
7  :- implementation.
8
9  main(!IO) :-
10     X = 1,
11     (
12         X = 2,
13         true
14     ;
15         X = 1,
16         loop(X),
17         io.write("X = ", !IO),
18         io.write(X, !IO),
19         io.nl(!IO)
20     ).
21
22 loop(X) :- loop(X). % illimited loop is considered
23                     % as erroneous code.
```